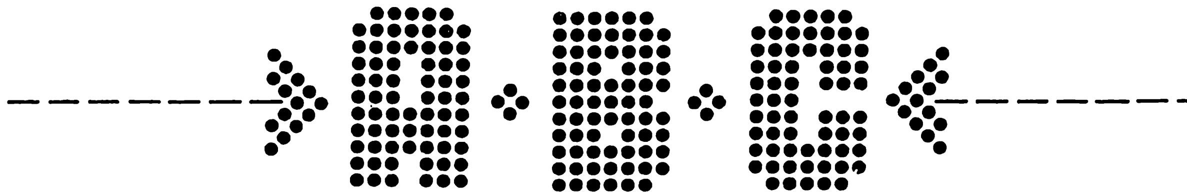
Roland Wacławek



asemblera

Roland Wacławek



asemblera

almapress

Studencka Oficyna Wydawnicza ZSP Warszawa 1989

Redaktor *Jacek Zyśk*

Redaktor techniczny *Włodzimierz Kukawski*

Projekt okładki i strony tytułowej *Andrzej Bilewicz*

Zdjęcie na okładce *Marek Nelken*

ISBN 83-7020-041-9

SPIS TREŚCI

1. WSTĘP

BASIC, PASCAL, C, FORTH, COBOL, LOGO... Długo można wyliczać listę tzw. języków programowania wysokiego poziomu. Programy w tych językach zapisujemy jako ciągi liter i innych symboli, używając często konwencji zapożyczonych z techniki a nawet życia codziennego. Czy to możliwe, że procesor — „serce" komputera — jest poliglotą, władającym tymi wszystkimi narzeczami?

Odpowiedź brzmi: nie! Komputer jest urządzeniem elektronicznym i jedynym językiem, jaki „rozumie" bezpośrednio, są sygnały elektryczne. Język impulsów, tzw. język maszynowy, jest „językiem narodowym" procesora. Aby procesor mógł wykonać program w innym języku, potrzebuje „tłumacza". Rolę tę spełniają translatory (programy tłumaczące) języków programowania. Zadaniem każdego z nich jest przekształcenie „zdań" języka wyższego poziomu na rozkazy języka maszynowego.

Człowiek myśli i porozumiewa się na co dzień w tzw. języku naturalnym, skrajnie odmiennym od języka procesora. Sensem istnienia języków programowania wyższego poziomu jest właśnie umożliwienie użytkownikowi formułowania programów w sposób możliwie zbliżony do potocznego. Człowiek piszący program nie widzi „twarzy" komputera, jest zwolniony od zgłębiania jego anatomii i „zwyczajów". Człowiek kontaktuje się tylko z pośrednikiem — translatorem — który instruuje procesor w jego języku wewnętrznym, co i jak zrobić.

Usługi tłumacza mają swoją cenę. Tłumaczenie zajmuje czas, zaś jego wynik, wiernie oddając sens oryginału (tzw. tekstu źródłowego), często jest tylko „niegramatycznym" zlepkiem rozkazów. Cóż, każdy język ma swą specyfikę i idiomy, które w przekładzie trzeba zastępować zagmatwanymi objaśnieniami i „przypisami". Wiedzą o tym znawcy literatury, studiujący języki obce tylko po to, aby przeczytać w oryginale Wergiliusza, Goethego lub Conrada. „Tyle razy żyjesz, ile znasz języków" — myśl ta odnosi się także do informatyki.

„Rozmawiając" z procesorem w jego „języku narodowym" możemy użyć wszystkich dostępnych w nim pojęć i sformułować nasze postulaty w sposób najbardziej zwięzły. Aby to osiągnąć, musimy jednak poznać bliżej procesor. Język to wszak nie tylko słownictwo, ale m.in. także składnia i stylistyka.

Programując w języku wewnętrznym można maksymalnie wykorzystać możliwości samego komputera i dołączonych do niego urządzeń. Z tego powodu programy, których efektywność (np. szybkość pracy) jest szczególnie istotna, pisze się właśnie w języku maszynowym. Programy stworzone w językach wyższego poziomu — BASIC, PASCAL itd. — często korzystają z usług podprogramów maszynowych, realizujących zadania szczególnie czasochłonne albo po prostu niemożliwe do zaprogramowania w tych językach.

Program w języku wewnętrznym jest ciągiem liczb zakodowanych w postaci elektrycznej. Liczby wyrażają konkretną czynność procesora (np. dodanie dwóch wartości) lub precyzują sposób jej wykonania. Posługiwanie się przy programowaniu kodami liczbowymi byłoby bardzo niewygodne. W praktyce rozkazy przedstawiamy w postaci symbolicznych

skrótów (np. ADD — dodaj). Tak zapisany program musi być przed wykonaniem przetłumaczony na kody liczbowe i wprowadzony do komputera. Wykonuje to program zwany asemblerem. Tłumaczenie przez asembler (asemblacja) różni się od pracy translatorów języków wysokiego poziomu tym, że każdej instrukcji symbolicznej odpowiada dokładnie jeden rozkaz maszynowy o ściśle określonym kodzie. Język asemblera, potocznie zwany po prostu asemblerem, nie jest więc odrębnym językiem programowania, lecz tylko wygodnym dla człowieka sposobem zapisu (notacji) rozkazów maszynowych.

Znajomość asemblera ma, poza „narzędziowym", jeszcze inny aspekt — światopoglądowy. Między programistą a procesorem nie ma już żadnego pośrednika uszczuplającego jego władzę nad maszyną. Dostępne stają się najbardziej nawet sekretne mechanizmy. Tajemniczy i „mądry" komputer stopniowo ulega odbrązowieniu.

Wreszcie po zgłębieniu ostatniego wątpliwego szczegółu, nadchodzi czas odkrywczej i zarazem oczyszczającej konstatacji: „procesor — cóż to za tępe liczydło!"

2. NIEZBĘDNE WIADOMOŚCI O SYSTEMIE DWÓJKOWYM

Poznając język maszynowy postaramy się nie wnikać niepotrzebnie w techniczne szczegóły pracy komputera. Jednym z wyjątków będzie dwójkowy (binarny) system liczbowy, stanowiący podstawę współczesnej techniki cyfrowej. Znajomość podstaw systemu dwójkowego jest bowiem podstawowym narzędziem w analizie i układaniu programów maszynowych.

2.1. Ile waży bit?

Termin „liczba" kojarzy się nam na ogół z jej zapisem w systemie dziesiętnym. Ta sama liczba może być jednak przedstawiona na wiele sposobów. Wybór reprezentacji dyktują warunki techniczne. System dziesiętny, stosowany powszechnie w nauce, technice i życiu codziennym, jest wygodny w „ręcznych" rachunkach. Urządzenia elektroniczne preferują jednak tzw. system dwójkowy.

Symbole używane do zapisu liczb nazywamy cyframi. W odróżnieniu np. od systemu rzymskiego, systemy: dwójkowy i dziesiętny są typu pozycyjnego. Znaczy to, że liczba cyfr jest z góry ograniczona, zaś znaczenie cyfry zależy nie tylko od niej samej, ale i od jej pozycji w stosunku do innych cyfr. W systemie pozycyjnym można przedstawić praktycznie każdą liczbę o skończonej wartości, zwiększając tylko liczbę pozycji użytych do jej zapisu.

Przy przechowywaniu liczb w urządzeniu technicznym korzystnie jest wartość każdej cyfry przedstawić jako stan pewnego zjawiska fizycznego, np. napięcia na kondensatorze. W systemie dziesiętnym (systemie o podstawie 10) należałoby dziesięciu cyfrom przypisać dziesięć różnych wartości napięcia. Rozpoznawanie cyfry, zakodowanej w formie napięcia, wymagałoby niezawodnego rozróżniania tych napięć, podobnie jak czyni to woltomierz cyfrowy. Jest to technicznie możliwe, lecz złożone. Łatwiej i pewniej jest używać tylko dwóch skrajnych stanów, np. 0 V i napięcia zbliżonego do pewnej wartości maksymalnej, przypuśćmy 5 V. Technologia wytwarzania elementów dwustanowych jest prosta i tania, co decyduje o opłacalności produkcji masowej.

Dysponujemy elementami dwustanowymi. Każdy z nich może zapamiętać jedną cyfrę dwójkową, czyli bit (ang. binary digit). Nie wnikając w szczegóły techniczne oba przeciwstawne stany oznaczymy umownie jako „0" (inaczej: L, low, poziom niski) i „1" (H, high, poziom wysoki). Te dwie cyfry umożliwiają przedstawienie w komputerze dowolnej informacji: liczb, tekstów, rysunków, a także programów. Bit o wartości 1 nazywamy często bitem ustawionym, bit o wartości 0 — skasowanym. System, w którym występują tylko dwie cyfry, nazywamy dwójkowym (systemem o podstawie 2). Jest on systemem naturalnym w zastosowaniach logicznych, gdzie także występują tylko dwie wartości: prawda (ang. true) i fałsz (ang. false).

Co oznacza w praktyce pozycyjność zapisu? Zacznijmy od liczby dziesiętnej:

```
1474 = (1*1000)+(4*100)+(7*10)+(4*1)
```

Współczynniki 1000, 100, 10 i 1 to tzw. wagi poszczególnych cyfr. Nie są one bynajmniej liczbami magicznymi, lecz kolejnymi potęgami podstawy systemu, czyli 10. Cyfry są numerowane od prawej do lewej, zaczynając od 0. Waga cyfry to podstawa systemu podniesiona do potęgi, odpowiadającej numerowi cyfry:

```
1474 =(1*10^3)+(4*10^2)+(7*10^1)+(4*10^0)
```

Reguły znane z systemu dziesiętnego występują także i w dwójkowym — z jednym wyjątkiem: podstawa systemu wynosi 2. Wagi bitów są potęgami nie 10, lecz 2. Aby wyznaczyć wagę bitu na danej pozycji, wystarczy podwoić wagę jego prawego sąsiada. W liczbie całkowitej pierwszy bit z prawej, zwany najmniej znaczącym albo najmłodszym, ma zawsze wagę 1. Ponieważ komputer posługuje się systemem dwójkowym, a ludzie — dziesiętnym, przekształcanie postaci zapisu liczb (tzw. konwersja systemów) z dwójkowego na dziesiętny i vice versa jest czynnością dość częstą. Aby znaleźć dziesiętny odpowiednik liczby dwójkowej, najprościej dokonać jej rozwinięcia:

```
1011 = (1*2^3)+(0*^2)+(1*2^1)+(1*2^0)
     =  1*8   + 0*4  + 1*2   + 1*1
     = 11 (dzieś.)
```

Co oznacza zapis 110: liczbę sto dziesięć w zapisie dziesiętnym, czy liczbę dwójkową o dziesiętnym odpowiedniku 6? Często stosowaną konwencją zapisu jest kończenie liczby dwójkowej literą „B", np. 110B. Niekiedy liczby dwójkowe poprzedza się znakiem %, np. %110 uznamy za liczbę dwójkową, 110 — za dziesiętną.

Liczbę dziesiętną najłatwiej przekształcić na dwójkową, odejmując od niej wagi kolejnych bitów. Oto one:

nr bitu	7	6	5	4	3	2	1	0
waga bitu	128	64	32	16	8	4	2	1

Odejmowanie zaczniemy od bitów najstarszych. Gdy odejmowanie okaże się niewykonalne (odjemnik przewyższa odjemną), to anulujemy je, zaś bitowi przypisujemy wartość 0. W przeciwnym razie różnica zajmie miejsce odjemnej, zaś bit przyjmie wartość 1. Jak wygląda liczba 53 w postaci dwójkowej?

```
         53    53    53    21     5     5     1     1

       -128   -64   -32   -16    -8    -4    -2    -1
       ----- ----- ----- ----- ----- ----- ----- -----
         --    --    21     5    --     1    --     0


bit =    0     0     1     1     0     1     0     1
```

Ostatecznie, dwójkową reprezentacją 53 jest 00110101B lub po prostu 110101B.

Pojedynczy bit rzadko wystarcza do przedstawienia jakiejś informacji. Bity łączy się zatem w grupy, zwykle po 8. Grupa ośmiu bitów, stanowiąca pewną jednostkę funkcjonalną i

uczestnicząca jako całość w podstawowych operacjach, nazywana jest bajtem (ang. byte). Przy pomocy jednego bitu można przedstawić tylko dwie liczby, 0 i 1. Dwa bity to już cztery liczby, od 00B (0) do 11B (3). Pełny bajt może reprezentować liczby od 00000000B (0) do 11111111B, czyli 255: w sumie 256 różnych wartości.

W nowoczesnych komputerach, a zwłaszcza mikrokomputerach, bajt jest podstawową „porcją" informacji. Z tego powodu pojemność pamięci podaje się nie w bitach, lecz w bajtach. Dla uniknięcia dużych liczb operuje się kilobajtem (1KB, 2↑10=1024 bajty) i megabajtem (1MB =1024 KB). Dla odróżnienia od powszechnie stosowanego „kilo", oznaczającego 1000, zapisując KB używamy dużego „K".

Chcąc zapamiętać wartości większe niż 255, trzeba połączyć kilka bajtów. Najczęściej używane są liczby dwubajtowe (szesnastobitowe), w systemach ośmio- i szesnastobitowych zwane często słowami (ang. word). Słowo pozwala przechować liczby od 0 do 65535.

2.2. Szczypta arytmetyki

Liczby dwójkowe dodajemy podobnie jak dziesiętne. Gdy po dodaniu dwóch cyfr uzyskujemy wartość niemożliwą do zapisania pojedynczą cyfrą, zachodzi tzw. przeniesienie. Odejmujemy wtedy od wyniku podstawę systemu, zaś do następnej, starszej pozycji dodajemy 1. W przypadku liczb dwójkowych przeniesienie wystąpi już wtedy, gdy wynik dodawania dwóch bitów będzie większy od 1:

```
 01110101
+00110110
---------
 10101011
```

Odejmowanie można zastąpić zmianą znaku odjemnika i dodawaniem. Jak przedstawić w systemie dwójkowym liczbę ujemną? Dotąd posługiwaliśmy się tylko całkowitymi liczbami dodatnimi. Liczby ujemne kojarzą się z pojęciem znaku. Jak wszystko inne, znak także trzeba zakodować dwójkowo. Jeden z bitów liczby, zwykle najstarszy, jest zarezerwowany dla znaku. Wartość 0 oznacza liczbę dodatnią, 1 — ujemną. Rozważając liczby jednobajtowe, zapis 00000011B oznacza 3, zaś 10000011B mógłby przedstawiać –3. Co jednak znaczą zapisy: 00000000B i 1000000B? Czy +0 i –0? Taka interpretacja jest naturalną konsekwencją przyjętej konwencji. Niestety, rodzi to poważne problemy techniczne. Dlatego w praktyce przyjął się inny system zapisu liczb ujemnych, mniej oczywisty, lecz nie zagrażający podwójną reprezentacją zera. Mowa o systemie uzupełnienia dwójkowego.

Aby wyznaczyć dwójkową reprezentację liczby ujemnej, należy wziąć jej dodatnią postać, zanegować wszystkie bity, a następnie dodać 1 do wyniku. Negacja bitu oznacza zmianę jego wartości na przeciwną (z 0 na 1 i 1 na 0). Oto jak wyznaczyć dwójkową postać liczby –40:

```
 00101000      +40
 11010111      negacja wszystkich bitów
+00000001      dodanie jedynki, czyli inkrementacja
---------
 11011000      -40 w systemie uzupełnienia dwójkowego
```

W identyczny sposób można zmienić na przeciwny, znak liczby ujemnej. Sprawdźmy:

```
11011000     -40 w systemie uzupełnienia dwójkowego
00100111     negacja wszystkich bitów
+00000001    inkrementacja
---------
00101000     +40
```

System uzupełnienia dwójkowego jest korzystny z technicznego punktu widzenia. Z odejmowania można zrezygnować, zastępując je prostą negacją i dodawaniem, zaś dodawanie liczb dodatnich i ujemnych jest prowadzone w myśl jednolitych reguł. Sprawdźmy:

```
 11011000    -40
+00101000    +40
---------    ---
 00000000      0
```

Ściśle biorąc, otrzymaliśmy 100000000, gdyż na najstarszej pozycji wystąpiło przeniesienie. Ponieważ jednak operujemy na bajtach, to dziewiąty bit już się „nie mieści" i jest odrzucany. Zachodzi tu analogia np. do licznika kilometrów w samochodzie, który po osiągnięciu stanu 99 999 wraca z powrotem do 00 000.

W systemie dopełnienia dwójkowego zachowana jest zasada, że najstarszy bit liczby ujemnej ma zawsze wartość 1, a dodatniej 0 (liczba 0 uchodzi za dodatnią). Rozważmy liczby jednobajtowe. O ile interpretacja liczb od 0B do 01111111B (0 do 127) jest jednoznaczna, to zapisy 10000000B do 11111111B mogą mieć dwa znaczenia: liczby dodatnie od 128 do 255 lub liczby ujemne od –128 do –1. W pierwszym przypadku mówimy o liczbach bez znaku (najstarszy bit służy do przedstawiania wartości liczby dodatniej), w drugim o liczbach ze znakiem. Wybór interpretacji należy do programisty. Na szczęście w większości operacji działania na liczbach bez znaku przebiegają identycznie jak na liczbach ze znakiem. Poza tym w zastosowaniach praktycznych wybór interpretacji narzuca się sam.

2.3. Podstawowe operacje logiczne

Podczas gdy w językach wysokiego poziomu dominują operacje arytmetyczne, programy maszynowe nasycone są tzw. operacjami logicznymi. W operacjach tych liczba dwójkowa traktowana jest nie jako całość, lecz jako zbiór pojedynczych bajtów.

Najprostszą, bo jednoargumentową operacją logiczną jest negacja logiczna (NOT). Polega ona na zmianie na przeciwną wartości wszystkich bitów. Bajt 10011101B po negacji logicznej przyjmie postać 01100010B. Negacja logiczna nazywana jest często dopełnieniem logicznym (ang. complement).

Spośród operacji dwuargumentowych najważniejsze są: suma logiczna (OR), iloczyn logiczny (AND) i różnica symetryczna (EXCLUSIV — OR, w skrócie: XOR). W każdym przypadku operacja odbywa się oddzielnie na parach odpowiadających sobie bitów w obydwu argumentach (operandach). Każdy bit wyniku zależy więc wyłącznie od stanu dwóch bitów w operandach o tym samym numerze (tej samej wadze).

W często stosowanej sumie logicznej dany bit wyniku jest zerem tylko wtedy, gdy od-

powiednie bity obydwu argumentów także są wyzerowane. W rezultacie iloczynu logicznego ustawione są tylko te bity, którym odpowiadają bity o wartości 1 w obydwu operandach. Przykłady:

```
      01010011              01010011
      00111001              11011001
OR    --------        AND   --------
      01111011              01010001
```

Suma logiczna jest używana do ustawiania wybranych bitów niezależnie od stanu pozostałych bitów liczby dwójkowej. Wystarczy wyznaczyć sumę logiczną tej liczby z inną liczbą, w której wartość 1 mają tylko bity na tych pozycjach, które zamierzamy ustawić w pierwszym operandzie. Iloczyn logiczny pozwala kasować wybrane bity (mówimy o „maskowaniu" bitów). Trzeba w tym celu poddać liczbę dwójkową operacji iloczynu logicznego z taką liczbą, w której ustawione są wszystkie bity z wyjątkiem przeznaczonych do skasowania. Oto przykłady. W pierwszym chodzi o ustawienie najstarszego bitu w pierwszym operandzie, w drugim — o skasowanie najmłodszego:

```
      00010110              10110111
      10000000              11111110
OR    --------        AND   --------
      10010110              10110110
```

W rezultacie różnicy symetrycznej wyzerowane są wszystkie bity, których odpowiedniki w argumentach mają identyczną wartość (0 i 0 lub 1 i 1). Tam, gdzie wartości bitów są różne, w wyniku występuje jedynka:

```
      01101011·
      01110110
XOR   --------
      00011101
```

Różnica symetryczna umożliwia selektywną negację wybranych bitów liczby dwójkowej. Wystarczy odpowiednio dobrać drugi argument, ustawiając w nim bity odpowiadające tym pozycjom pierwszego argumentu, które mają zostać zanegowane. Jeżeli drugi argument jest zerem, rezultat jest równy pierwszemu argumentowi. Gdyby w drugim argumencie były ustawione wszystkie bity, operacja XOR odpowiadałaby jego negacji logicznej.

Operacje logiczne na dwójkowych postaciach liczb są dostępne w niektórych językach programowania wysokiego poziomu, m.in. w MICOROSFT—BASIC. Występujące w tym języku zmienne typu całkowitego (np. I%, WART%) są przechowywane w pamięci jako dwubajtowe liczby dwójkowe ze znakiem. Jeśli dwie liczby połączone są operatorem logicznym OR, AND lub XOR, odpowiada to operacji logicznej na dwójkowych, szesnastobitowych postaciach argumentów. Podobnie umieszczenie operatora (symbolu operacji) NOT przed liczbą odpowiada negacji logicznej jej dwójkowej reprezentacji. Łatwo teraz pojąć, dlaczego NOT 0 = –1, a nie 1, co wydawałoby się bardziej logiczne. Uwzględniając szesnastobitową postać liczby, NOT 0 = NOT 0000000 0000000B = 1111111 1111111B, co w systemie uzupełnienia dwójkowego odpowiada właśnie dziesiętnemu –1.

2.4. System szesnastkowy

Aczkolwiek system dwójkowy wiernie odzwierciedla sposób przedstawienia liczb w pamięci komputera, to rozwlekłość zapisu nie czyni go wygodnym w użyciu. Programując w języku wewnętrznym nie wolno jednak tracić z oka kompozycji bitów w poszczególnych bajtach. Zachowajmy więc system dwójkowy, stosując tylko zwięźlejszą formę zapisu.

Zacznijmy od podziału liczby dwójkowej na czterobitowe grupy, poczynając od prawej. Grupy takie często noszą nazwę półbajtów (ang. nibble). Każdy z półbajtów wyrazimy za pomocą pojedynczego symbolu. W ten sposób zmniejszymy czterokrotnie ilość cyfr, zachowując łatwy dostęp do poszczególnych bitów, czego nie daje system dziesiętny.

Jeden półbajt może przedstawić liczby dwójkowe od 0000B do 1111B (0 do 15). Pierwszych dziesięć kombinacji można wyrazić cyframi dziesiętnymi „0" do „9". Dla pozostałych sześciu cyfry trzeba wymyślić. Użyjmy pierwszych 6 liter alfabetu „A" do „F". W ten sposób stworzymy system o podstawie 16, zwany szesnastkowym. Rozpowszechniona jest też niezbyt poprawna nazwa: system heksadecymalny. Oto cyfry szesnastkowe i ich wartości:

Cyfra szesnastkowa	Wartosc dwojkowa	Wartosc dziesietna
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Przekształćmy liczby dwójkowe do postaci szesnastkowej:

```
11100010B          0011101011110001B
1110 0010          0011 1011 1111 0001
 E    2             3    B    F    1
```

Konwersja z zapisu szesnastkowego na dwójkowy jest równie prosta. Wystarczy zamiast każdej z cyfr szesnastkowych wstawić odpowiadającą cyfrze grupę czterech bitów:

```
3C = 00111100B        F0A7 = 1111000010100111B
```

Liczby szesnastkowe zapisujemy zazwyczaj dodając na końcu literę H, np. 3CH lub OFOA7H (pierwszym znakiem musi być cyfra 0-9). Niekiedy stosowana jest inna konwencja, w której liczba szasnastkowa poprzedzona jest znakiem „#", np. # 3C lub # FOA7 (czasem zamiast „#" używa się „$", zwłaszcza w asemblerze mikroprocesora 6502). Zauważmy, iż przejście od systemu dwójkowego do szesnastkowego i na odwrót jest łatwe i naturalne — inaczej niż w przypadku systemu dziesiętnego. Wynika to z faktu, że liczba 16 jest potęgą liczby 2. System szesnastkowy ma charakter pozycyjny i obowiązują w nim znane reguły:

```
3BF1H = (3*16^3)+(11*16^2)+(15*16^1)+(1*16^0)
      = (3*4096)+(11*256) +(15*16)   +(1*1)
      = 15345
```

Cyfr szesnastkowych i odpowiadających im kombinacji bitów warto po prostu się nauczyć. Ten niewielki wysiłek wkrótce się opłaci.

Ułatwieniem w przeliczaniu liczb jednobajtowych z systemu dwójkowego i szesnastkowego na dziesiętny i odwrotnie jest tablica w Dodatku A.

3. JĘZYK MASZYNOWY I JĘZYK ASEMBLERA

Choć ogólne zasady działania mikroprocesorów są zbliżone, poszczególne ich typy różnią się istotnie szczegółami konstrukcyjnymi. Wpływa to z kolei na sposób programowania. Każdy typ mikroprocesora ma inną budowę i zestaw dostępnych operacji oraz własny język wewnętrzny. Szczególnie popularne są w Polsce dwa procesory: INTEL 8080 i ZILOG Z80. 8080 był pierwszym mikroprocesorem ośmiobitowym, który doczekał się wielkiego rozpowszechnienia. Jego odpowiednik (MCY880) jest produkowany także w Polsce. W porównaniu z 8080 procesor Z80 jest wygodniejszy w zastosowaniu i posiada większe możliwości. Twórcy Z80 postarali się jednak, aby mógł on wykonywać programy przeznaczone dla 8080 (z drobnymi wyjątkami). Obydwa procesory mają podobną organizację, a technika ich programowania jest zbliżona.

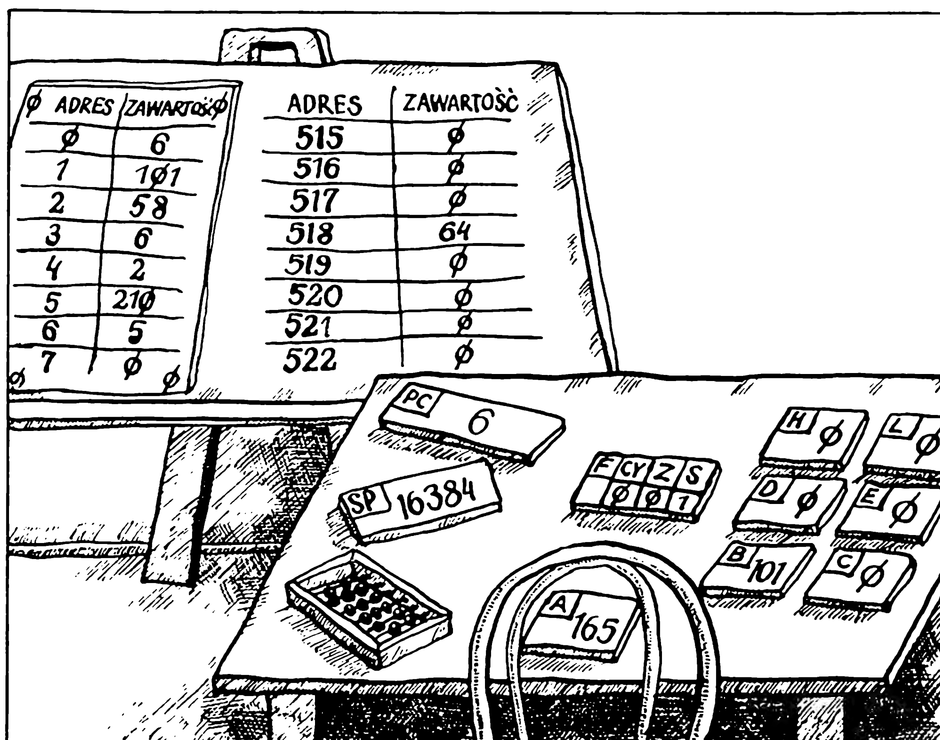
Podczas gdy 8080 stosowany jest głównie w systemach przemysłowych, w komputerach osobistych, a zwłaszcza domowych, dominuje Z80. Procesor ten jest „sercem" takich mikrokomputerów jak ZX81 i ZX Spectrum, MERITUM, Laser 110/210/310, CPC 464/664/6128 i Joyce, PCW 8256 oraz wszystkich systemów MSX. Procesory 8080 i Z80 umożliwiają też pracę w najpopularniejszym wśród ośmiobitowców systemie operacyjnym CP/M 80. Z tego powodu w dodatkowy procesor Z80 wyposażane są maszyny skonstruowane pod kątem innych mikroprocesorów, np. Apple II i Commodore C-128.

Zajmiemy się programowaniem procesora Z80. Ograniczymy się przy tym do pewnego podzbioru jego możliwości, odpowiadającego praktycznie procesorowi 8080. Wystarczy nam to w zupełności do tworzenia nawet dość złożonych programów bez obciążania się nadmierną ilością szczegółów. Lepiej opanować solidnie podstawy programistycznego rzemiosła stosując niezbędne minimum „narzędzi". Gdy zdobędziemy nieco wyczucia i doświadczenia, szybko opanujemy bardziej zaawansowane mechanizmy i nauczymy się je sensownie stosować. Uzupełnieniu wiadomości o procesorach 8080 i Z80 poświęcony jest ostatni rozdział.

Aby programować procesor w jego języku wewnętrznym, trzeba najpierw poznać sposób jego działania, jakże odmienny od funkcjonowania języków wysokiego poziomu.

3.1. Ćwiczenie dla wyobraźni

Przed nami tablica. Jej powierzchnia podzielona jest na sieć niewielkich pól-komórek. Każde pole ma swój numer: 0, 1, 2 itd. W jednej komórce wolno zapisać tylko jedną liczbę, o wartości od 0 do 255, lub, jeśli wolimy, od –128 do 127. Odpowiada to jednobajtowej liczbie dwójkowej, prawda? Numer komórki nazywamy jej adresem. Najmniejszy adres wynosi zawsze 0, zaś największy zależy tylko od rozmiaru tablicy. W naszym przypadku wynosi on 65535, co odpowiada 65536 komórkom.



Pod łokciami czujemy blat biurka. Leży na nim liczydło i kilka tabliczek. Siedem mniejszych jest oznaczonych literami A, B, C, D, E, H, L i ma rozmiar kratek tablicy (może pomieścić liczbę jednobajtową). Tabliczka A zajmuje uprzywilejowane miejsce na wprost naszego nosa, zaś reszta tabliczek jest powiązana w pary: B z C, D z E i H z L. W razie czego tabliczki pary można zestawiać i zapisywać na nich liczby dwubajtowe. W tym przypadku B, D i H leżą zawsze z lewej i służą do zapisu starszych cyfr. Ósma mała tabliczka, nazwana F, podzielona jest na trzy pola. W każdym z nich można zapisać tylko jeden bit. Oprócz tego na blacie leżą dwie duże tabliczki, nazwane PC i SP. Można na nich zapisać po jednej liczbie dwubajtowej (0-65535). Ostatnim rekwizytem jest lista rozkazów. Zawiera ona liczbowe kody poszczególnych czynności oraz dokładny opis reguł ich wykonania.

Przedstawiona rupieciarnia jest modelem komputera. Tablica to pamięć operacyjna (PAO), zaś my wraz z biurkiem reprezentujemy procesor (CPU). Będziemy sterować działaniem modelu.

Zadaniem pamięci operacyjnej jest przechowywanie niezbędnych informacji, czyli danych i rozkazów programu. Procesor przetwarza dane ściśle według wskazówek programu. Przyjrzyjmy się tablicy. Jej część przesłania szyba. Komórki pod szybą można łatwo odczytać, lecz nie sposób zmienić ich zawartości. Odpowiada to pamięci typu ROM (tylko do odczytu). Reszta tablicy jest dostępna dla gąbki i kredy. Można zetrzeć starą zawartość komórki i w jej miejsce wpisać nową liczbę. To pamięć RAM (zapis/odczyt). Tabliczki na biurku to rejestry wewnętrzne. Przypominają one komórki tablicy, ale są „pod ręką", służąc do roboczych notatek przy wykonywaniu działań. Na tabliczce A notujemy wynik większości operacji. Podwójna tabliczka (dwubajtowy rejestr) PC zawiera numer komórki z opisem następnej

czynności do wykonania (adres kolejnego rozkazu). PC nazywany jest w związku z tym wskaźnikiem programu lub częściej, lecz mniej precyzyjnie, licznikiem programu (ang. program counter).

Tablica jest już zapisana. Zacznijmy działać. Rzut oka na PC: zawiera adres 0. Skoro PC wskazuje komórkę 0, trzeba przyjąć, że jej zawartość jest rozkazem. Z komórki 0 odczytaliśmy liczbę 6. Spojrzenie na listę rozkazów: „Do rejestru B wpisz liczbę podaną w następnej komórce pamięci. Rozkaz zajmuje dwie komórki". Pierwszą czynnością jest powiększenie PC o ilość komórek PAO zajętych przez rozkaz. Od tej chwili PC zawiera 2. Teraz posłusznie przepiszemy odczytaną liczbę 101 do B. Rozkaz wykonany, wszystko zaczyna się od początku.

Spoglądamy na PC: zawiera 2. Pod adresem 2 mamy liczbę 58, czyli kod „Do A wpisz liczbę z tablicy (z PAO) o adresie podanym w dwóch następnych komórkach. Długość rozkazu: 3 komórki". Przede wszystkim zwiększamy PC o 3 (od tej pory PC zawiera 5). Adres może mieć wartość od 0 do 65535, nie można więc zmieścić go w pojedynczej komórce. Potrzebne jest dwubajtowe słowo. Bezpośrednio po kodzie rozkazu następuje młodszy bajt adresu, za nim starszy. Taka kolejność zapisu bajtów słów jest żelazną regułą. Waga starszego bajtu przewyższa 256 razy wagę młodszego. Odczytujemy obydwie komórki, wyznaczamy adres (256*2+6=518) i kopiujemy zawartość komórki nr 525 do rejestru A. Odtąd A zawiera 64.

Kolejny rozkaz: komórka nr 5 przechowuje kod 128 — „Do zawartości akumulatora dodaj liczbę z rejestru B i zasygnalizuj cechy wyniku operacji ustawiając wskaźniki stanu. Długość rozkazu: 1 bajt". W akumulatorze mamy 64, w rejestrze B — 101, suma wynosi 165. Wpisujemy ją do akumulatora zamiast pierwszego składnika. Rejestr B zachował swą wartość. Gotowe? Prawie. Pozostało nadać właściwą wartość wskaźnikom (bitom) stanu.

Trzy wskaźniki stanu (flagi) to jednobitowe pola rejestru F, zwanego rejestrem stanu lub rejestrem flagowym, w których większość operacji odnotowuje charakterystyczne cechy rezultatu. Bit Z (ZERO) wskazuje, czy wynik działania był zerem. Jeżeli tak, bit przyjmuje wartość 1. Gdy wynik operacji jest różny od zera, bit Z jest zerowany. Bit S (SIGN=znak) określa, czy wynik jest liczbą ujemną w sensie arytmetyki uzupełnieniowej. Innymi słowy, wskaźnik S przyjmuje tę samą wartość, co najstarszy bit akumulatora (bit nr 7): 1 dla liczb ujemnych, 0 dla dodatnich. Jeżeli wynik operacji interpretujemy jako liczbę bez znaku, to wskaźnik S podaje, czy wynik jest większy od 127. Trzeci wskaźnik jest oznaczany jako C (CARRY) i sygnalizuje przeniesienie. Aby uniknąć pomyłek wskutek identycznego oznaczenia bitu przeniesienia i jednego z rejestrów, bit CARRY będziemy zapisywać jako CY. Przy dodawaniu i odejmowaniu bit CY jest jakby lewostronnym przedłużeniem akumulatora, jego dziewiątym bitem. Jeśli suma dwóch składników (traktowanych jako liczby bez znaku) wypada większa od 255, to bit CY zostanie ustawiony, w przeciwnym razie — wyzerowany.

Nasz wynik nie jest zerem (w pole Z tabliczki F wpisujemy 0), jednak wynik jest większy od 128 (165 = 10100101B). W pole S musimy więc wstawić 1. Przeniesienie nie zaszło, więc bit CARRY zerujemy. Pozostaje zwiększyć PC o długość rozkazu, czyli o 1. PC zawiera 6, rozkaz wykonany.

W komórce 6 mamy kod 210: „Jeżeli wskaźnik CY ma wartość 0, wpisz do rejestru PC dwubajtową liczbę zawartą w dwóch kolejnych komórkach. W przeciwnym razie nic nie rób. Długość rozkazu: 3 bajty". Zwiększamy PC o 3 i przyglądamy się bitowi CY. Jest wyzerowany. Zgodnie z poleceniem wpisujemy do PC bajty 5 i 0, przedstawiające dwubajtową liczbę 5. Koniec.

Wpisanie do PC nowej zawartości oznacza zaburzenie normalnej kolejności wykonywania programu, czyli skok. Następny rozkaz zostanie pobrany spod adresu wskazanego w rozkazie skoku. W naszym przypadku skok był warunkowy, gdyż mógł zostać wykonany albo nie, zależnie od okoliczności (w języku BASIC: IF... THEN GOTO).

Po wykonaniu skoku PC zawiera 5. Ten rozkaz już wykonywaliśmy, mamy więc do czynienia z pętlą programową w języku maszynowym. Zwiększamy PC o 1 i powtarzamy dodawanie: 165+101. Suma składników wynosi 266, czyli dwójkowo 100001010B. W akumula-

torze mieści się tylko osiem młodszych bitów (00001010B, dziesiętnie: 10), zaś dziewiąty, najstarszy, przechodzi do wskaźnika CY. Wskaźniki Z i S zerujemy i przechodzimy do następnego rozkazu pod adresem 6. Tym razem warunek wykonania skoku nie jest spełniony (CY=1), zawartość licznika programu PC pozostaje niezmieniona. Następny kod rozkazu odczytamy zatem z komórki nr 9.

Wystarczy. W przyszłości realizację programu zlecimy procesorowi. Zabawa w procesor miała tylko zilustrować mechaniczny, bezmyślny tryb realizacji programu maszynowego. Zauważmy, że operacje były bardzo prymitywne i nie pozostawiały wykonawcy żadnej swobody interpretacji. Prócz tego w opisie czynności nie było sformułowań typu: „weź wynik przedostatniej operacji", „jeśli przy wykonaniu ostatniego rozkazu...", itd. Procesor nie pamięta historii swych działań i każdy rozkaz traktuje w całkowitym oderwaniu od poprzednich. Jeśli między poszczególnymi fragmentami programu ma być przekazana jakakolwiek informacja, może się to odbyć wyłącznie za pośrednictwem komórek PAO lub rejestrów, w tym rejestru stanu F z bitami wskaźników.

Wskaźniki zmieniają swój stan wyłącznie podczas operacji „przetwórczych", jak dodawanie, odejmowanie lub suma logiczna. Przesyłanie danych z udziałem rejestrów i komórek PAO nie wpływa na bity stanu. Stan wskaźników nie jest więc tylko zwykłym odzwierciedleniem zawartości akumulatora.

Zarówno program, jak i dane przedstawione są w pamięci operacyjnej jako masa bajtów. Sposób interpretacji każdego bajtu zależy tylko od programu. Co więcej, może zaistnieć sytuacja, w której komórka PAO będzie raz interpretowana jako rozkaz, innym razem jako dana. Wpiszmy do komórki nr 4 naszej „tablicy" zamiast 0 — 2 i uruchommy program ponownie. Drugi rozkaz załaduje do akumulatora zawartość komórki nr 6, czyli liczbę 210. Ta sama komórka zostanie w chwilę później zinterpretowana jako kod rozkazu skoku! Może zajść sytuacja, że program maszynowy zapisze nową wartość do którejś z komórek kodu i w ten sposób w trakcie pracy zmodyfikuje sam siebie. Procesor po prostu nie odróżnia (bo i po czym?) bajtów kodu programu od bajtów danych. Wszystkie bajty, na które kiedykolwiek wskaże licznik programu PC, będą potraktowane jako rozkazy — oto jedyna reguła.

Omówione cechy programu maszynowego zdecydowanie odbiegają od właściwości programów w językach wyższego poziomu, jak BASIC lub PASCAL. Program i dane są rozróżniane w sposób naturalny, a sam translator języka chroni nas przed wielu nonsensownymi operacjami, jak np. próba zapamiętania zamiast liczby, imienia kolegi. Programy w języku BASIC, mimo zróżnicowania dialektów, można zaadaptować do pracy na różnych komputerach. W języku maszynowym jest inaczej. Każdy typ procesora posiada własny język wewnętrzny. Liczba 133 (85H) interpretowana jako rozkaz przez procesor Z80, oznacza „do akumulatora dodaj zawartość rejestru L". Procesor MOS 6510 rozpozna tę samą liczbę jako polecenie zapamiętania zawartości akumulatora we wskazanej komórce PAO.

Język maszynowy daje programiście zupełną swobodę korzystania z procesora i PAO. Procesor zrealizuje posłusznie każde życzenie, które da się wyrazić w formie programu — nawet życzenie absurdalne. Całkowita odpowiedzialność za pracę komputera leży więc po stronie programisty. Ceną swobody jest też słabość mechanizmów wczesnego wykrywania błędów. Jeśli przy wykonywaniu programu w języku wyższego poziomu wystąpi błąd, operator otrzyma najczęściej komunikat o jego lokalizacji i charakterze. Usterka w programie maszynowym kończy się zwykle „załamaniem" systemu, czyli utratą możliwości porozumiewania z komputerem. Zawartość pamięci RAM jest przy tym często niszczona. W przypadku takiego „zacierania śladów" wyśledzenie przyczyny błędu może być uciążliwe. Przy tworzeniu programów maszynowych jest więc niezbędna szczególna staranność i systematyczność.

3.2. Co to jest asembler?

Bawiąc się w procesor założyliśmy, że program i dane znajdują się już w PAO, nie wnikając jednak, skąd się tam wzięły. Starsze komputery dysponowały bateriami przełączników, pozwalających zapisywać wybrane komórki PAO bez udziału procesora, a następnie uruchamiać program od wybranego adresu. W mikrokomputerach byłoby to zbędną komplikacją.

Po włączeniu zasilania licznik PC mikroprocesora Z80 jest zerowany, a więc mikroprocesor rozpoczyna wykonywanie programu od komórki 0. Pod tym adresem znajduje się pamięć ROM z zapisanym w niej na stałe programem maszynowym. Może być to tzw. bootstrap, czyli program ładujący z dyskietki np. system operacyjny. W komputerach domowych występuje jednak zazwyczaj interpreter języka BASIC. W każdym razie po włączeniu mikrokomputer jest w stanie nawiązać z użytkownikiem dialog za pośrednictwem klawiatury i monitora.

W języku BASIC zapis i odczyt komórek PAO jest możliwy dzięki instrukcji POKE i funkcji PEEK. POKE aaaa, nn zapisuje do komórki o adresie aaaa bajt nn, funkcja PEEK (aaaa) odczytuje bajt z komórki numer aaaa. Samo wprowadzanie programu maszynowego do PAO nie jest problemem. Znacznie trudniej jest ten program ułożyć. Wszystkie czynności należy bowiem opisać w formie jednobajtowych liczb, umieszczonych w ściśle określonych komórkach PAO. Niezbędna jest oczywiście znajomość listy rozkazów. Trzeba wiedzieć, jak komponować prymitywne operacje, żeby osiągnąć pożądany skutek. Wiadomości te można jednak posiąść, chociażby z książek. Sam algorytm, a nawet precyzyjna koncepcja zrealizowania go dostępnymi rozkazami maszynowymi nie wystarcza. Konieczne jest jeszcze zakodowanie tych wszystkich rozkazów w postaci ciągu bajtów.

Zapisywanie programów maszynowych wprost w postaci kodów liczbowych jest w praktyce nierealne. Program taki byłby zupełnie nieczytelny. Gdyby jeszcze program składał się z samych kodów operacji! Wiemy jednak, że część rozkazów ma jedno- lub dwubajtowe argumenty. Jeśli argument ten przedstawia adres, np. w rozkazie skoku, powstają nowe komplikacje. Trzeba bowiem znać wszystkie adresy, do których się skacze, zapisuje dane itd. Przypuśćmy, że należy wykonać skok „w przód", do niezakodowanego jeszcze fragmentu programu. Jak określić adres skoku? Należy zarezerwować dwa bajty dla argumentu, zaś na zakończenie uzupełnić brakujące adresy. Wszystko to jest uciążliwe i tworzy mnóstwo okazji do popełnienia przypadkowych błędów.

Prawdziwa bieda zaczyna się przy wprowadzaniu poprawek. Niech zajdzie konieczność dodania jednego, jedynego rozkazu w środku programu. Aby zrobić dla niego miejsce, konieczne jest przesunięcie wszystkich następujących po nim rozkazów. Rozkazy te zmieniają swoje adresy. Do wielu z nich z pewnością zaplanowano skoki. Adresy zapisane w rozkazach skoków stały się nieaktualne. W praktyce oznacza to konieczność przeanalizowania całego programu i aktualizacji wszystkich zmienionych adresów w argumentach skoków. Uruchomiając nieduży program maszynowy trzeba nieraz wprowadzić kilkadziesiąt modyfikacji. Brrr!!!

Ratunek tkwi w zastosowaniu języka symbolicznego. Zamiast kodować rozkazy wprost w formie liczb, użyjemy mnemonicznych nazw, związanych z funkcją rozkazu. W miejsce kodu 128 (do akumulatora dodaj rejestr B) napiszemy ADD A, B. ADD oznacza po angielsku dodaj, zaś LD pochodzi od angielskiego słowa LOAD (ładuj). Rejestry możemy podać jawnie, zaś argumenty będące stałymi liczbowymi zapisać w systemie dziesiętnym. Każdy rozkaz umieszczamy w oddzielnym wierszu, niezależnie od tego, z ilu bajtów się on składa. Tak wyglądałby w postaci symbolicznej nasz mini-program, analizowany w poprzednim rozdziale:

```
0       LD   B,  101
2       LD   A,  (518)
5       ADD  A,  B
6       JP   NC, 5
```

W czterech wierszach zapisano kolejno: „ładuj do B stałą 101", „ładuj do A bajt z komórki 518", „dodaj B do A", „jeśli CY=0, skocz do adresu 5". Liczby z lewej strony wskazują adresy rozkazów. Ponieważ długość każdego rozkazu jest stała, w oparciu o adres pierwszego można wyznaczyć położenie wszystkich następnych. Znajomość adresów była niezbędna do podania argumentów skoków. Jak uniknąć bezpośredniego podawania adresów? Stosując tzw. etykiety (ang. label). Oto nasz program w nowej wersji:

```
        LD   B,  101
        LD   A,  (518)
DODAJB: ADD  A,  B
        JP   NC, DODAJB
```

Przed wszystkimi rozkazami lub innymi komórkami PAO, do których chcemy odwołać się w programie, umieszczamy etykiety (np. DODAJB). Przyjmijmy, że nazwa etykiety może składać się z 1 do 6 znaków alfanumerycznych (liter lub cyfr) i że pierwszy znak musi być literą. Nazwy: „ALFA", „X1", „Q" i „BE3K" są dozwolone, „1X" lub „TYP-2" — nie. Czasem żąda się, aby etykieta nie była nazwą rozkazu ani rejestru. Etykieta musi znajdować się na początku wiersza i na ogół powinna być oddzielona dwukropkiem. Konstruując program, nie musimy znać jego przyszłej lokalizacji w PAO. W tej fazie interesuje nas tylko prawidłowy układ rozkazów i związki funkcjonalne między nimi. Planując np. skoki, nie podamy wprost adresu, lecz tylko nazwę etykiety, umieszczonej przed rozkazem, stanowiącym cel skoku. Wybór nazwy jest przywilejem programisty. Warto wybierać nazwy mnemoniczne związane z funkcją danego fragmentu programu (DODAJ, KASUJ, WYDRUK). Ułatwia to istotnie orientację w programie.

Użycie symbolicznych nazw rozkazów i etykiet ułatwiło zapis programu, lecz pozostał problem tłumaczenia go na kod liczbowy. Czynność tę może jednak wykonać maszyna. Zamiast do zeszytu, wpiszemy program symboliczny do pamięci komputera, a przetłumaczenie go na postać maszynową zlecimy innemu programowi. Program tłumaczący zapis symboliczny na kod maszynowy nazywamy asemblerem. Język symboliczny jest przeto nazywany często językiem asemblera. Zdarza się, że zamiast „język asemblera" używane jest po prostu określenie „asembler". Słowo to ma więc w żargonie informatyków dwa znaczenia: język symboliczny i program tłumaczący ten język na postać dwójkową.

Firmy INTEL i ZILOG forsowały dla swych procesorów różne języki symboliczne. Znaczy to, że ten sam program może mieć inną postać w obydwu językach. Do tej pory posługiwaliśmy się zapisem asemblerowym dla Z80, który jest nowocześniejszy i bardziej konsekwentny. Pozostaniemy przy nim także w następnych rozdziałach. Nic nie stoi na przeszkodzie, aby w języku symbolicznym Z80 zapisywać programy także i dla 8080, pod warunkiem, że ograniczymy się do rozkazów akceptowanych przez ten procesor. W końcu, niezależnie od sposobu zapisu, uzyskamy identyczny kod maszynowy. Oto jak wyglądałby nasz pierwszy programik w notacji 8080:

```
        MVI  B,  101
        LDA      518
DODAJB: ADD  B
        JNC      DODAJB
```

Problem w tym, że wiele asemblerów, zwłaszcza w stystemie CP/M 80, dopuszcza jedynie zapis symboliczny firmy INTEL. W związku z tym w Dodatku B zamieszczono oprócz symbolicznych nazw rozkazów dla Z80 także równoważny im zapis w konwencji INTEL.

3.3. Zapis programów w języku asemblera

W przyszłości będziemy zapisywać programy maszynowe wyłącznie w postaci symbolicznej, pozostawiając asemblerowi ich tłumaczenie i rozlokowanie w pamięci. Poznajmy zatem podstawowe zasady zapisu programów w tym języku. Oto przykładowy program w języku asemblera:

```
; Przykład programu w języku asemblera
BETA     EQU  50           ;definicja stałej o nazwie BETA
         ORG  25600        ;lokalizacja programu w PAO
START:   LD   A, (ALFA)    ;ładuj do A bajt z komórki ALFA
         LD   B, BETA      ;ładuj do B stałą BETA
         ADD  A, B
         JP   NC,PRZEN
         LD   B, BETA+5    ;wpisz do B stałą BETA+5
PRZEN:   CPL               ;wyznacz dopełnienie dwójkowe A
         JP       START
ALFA:    DEFB 0            ;rezerwacja bajtu (war. pocz=0)
APRZEN:  DEFW PRZEN-1      ;rezerwacja słowa (w. =PRZEN-1)
```

Program ma typową budowę, każdy wiersz zawiera pojedynczą instrukcję języka symbolicznego. Instrukcja dzieli się na cztery pola (niektóre z pól mogą być puste). Z lewej strony występuje pole etykiety, w środku pola operacji i argumentów, z prawej, oddzielone średnikiem, pole komentarza. W większości wierszy pole etykiety jest puste. Sensownie jest bowiem umieszczać etykiety tylko przed tymi punktami programu, do których zamierzamy się odwoływać w innych miejscach. Pole komentarza rozpoczyna się średnikiem, po którym można umieścić dowolny tekst (jak po REM w języku BASIC). Tekst ten jest przeznaczony dla osoby analizującej treść programu. Ponieważ tekst komentarza jest ignorowany podczas asemblacji, nie ma on najmniejszego wpływu na wynikową (dwójkową) postać programu (w języku BASIC jest inaczej: komentarz nie wpływa na sposób działania programu, lecz zwiększa jego objętość i zwalnia pracę). Stosowanie komentarzy jest bardzo wskazane. Należy je wykorzystywać do objaśniania sensu poszczególnych operacji, znaczenia komórek pamięci itd. Gdy potrzebne są obszerne opisy, można użyć w programie linii zawierających wyłącznie pole komentarza.

W polu operacji najczęściej występuje nazwa rozkazu maszynowego. Niektóre rozkazy nie potrzebują argumentów (np. CPL). Inne wymagają jednego argumentu (np. JP START) lub więcej (np. ADD A, B). Jeśli argumentów jest kilka, oddzielamy je przecinkami.

Niektóre instrukcje mogą zawierać nie rozkazy maszynowe, lecz tzw. dyrektywy. Dyrektywa nie definiuje żadnego rozkazu maszynowego i często nie dostarcza żadnego kodu. Dyrektywy są bowiem wskazówkami dla asemblera, określającymi sposób tłumaczenia programu lub interpretację nazw.

Dyrektywą występującą praktycznie w każdym programie jest ORG. Ma ona jeden argument, określający adres, pod którym zostanie umieszczony w PAO kod maszynowy, wytworzony przez instrukcje następujące po ORG. W naszym przykładzie rozkaz: LD A, (ALFA) zostanie wpisany pod adres 25600, zaś następne rozkazy — bezpośrednio za nim. W razie potrzeby można użyć kilku dyrektyw ORG w różnych punktach programu. Pozwala to umieścić dane lub fragmenty kodu w różnych obszarach PAO.

Poznaliśmy już jeden sposób przypisania wartości nazwie przez umieszczenie jej w polu etykiety wybranej instrukcji. Inny sposób to użycie dyrektywy EQU. Nazwie wymienionej po lewej stronie słowa EQU zostanie przypisana wartość podana po prawej. W naszym przykładzie nazwie BETA nadano wartość 50. Od tej pory przy każdym wystąpieniu w programie nazwy BETA zostanie zamiast niej wstawiona wartość 50. Operowanie nazwami sta-

łych zamiast bezpośrednio ich wartościami ma liczne zalety. Po pierwsze, można nazwać stałą w sposób objaśniający jej funkcję (np. CZAS EQU 30). Po drugie, zdarza się, że ta sama stała występuje w wielu miejscach programu. Gdy stałą podaliśmy wprost, przy każdej zmianie wartości stałej trzeba modyfikować wszystkie instrukcje, w której została użyta. Inaczej przy zastosowaniu nazwy stałej. Wartość jest przypisywana nazwie w jednym miejscu programu, na ogół na początku. Wystarczy wtedy zmienić tylko jedną dyrektywę EQU.

Program to nie tylko sam kod maszynowy, ale najczęściej i zbiór komórek PAO, przechowujących dane, pośredniczących w ich przekazywaniu itp. Do rezerwacji jednobajtowych komórek służy dyrektywa DEFB (w asemblerach dla 8080: DB). Po słowie DEFB należy wymienić jeden lub więcej argumentów. Każdy z nich powinien być liczbą o wartości od 0 do 255 (liczne asemblery dopuszczają zakres od –255 do 255; –1 znaczy tyle samo co +255, –255 co +1). Przy tłumaczeniu programu asembler zarezerwuje tyle kolejnych komórek PAO, ile podano argumentów DEFB. Do każdej z komórek zostanie wpisana podana wartość.

Programy zawierają często teksty, złożone ze znaków w kodzie ASCII. Wpisywanie tekstów bezpośrednio w postaci ciągu kodów byłoby uciążliwe. Dlatego dyrektywa DB dopuszcza jeszcze inną postać argumentu ciąg złożony z jednego lub więcej znaków, ujęty w apostrofy:

```
DB 'A', 0, 'TEKST 1'
```

Zamiast znaków, asembler podczas tłumaczenia programu wstawi w ich miejsce odpowiednie kody ASCII. Pojedynczy znak w apostrofach może wystąpić także jako argument rozkazu, w roli stałej liczbowej reprezentującej kod wskazanego znaku. Poniższe zapisy są równoważne (kod ASCII dla litery „A" =65):

```
LD  B, 65
LD  B, 'A'
```

Często zachodzi konieczność rezerwacji PAO dla liczb dwubajtowych. Umożliwia to dyrektywa DEFW (dla 8080: DW). Po DEFW musi wystąpić jeden lub kilka argumentów o wartościach od 0 do 65535 (czasem od –65535 do 65535). W tym przypadku dla każdego argumentu zostaną zarezerwowane dwie kolejne komórki PAO. W komórce o niższym adresie wpisany zostanie młodszy bajt wartości, w następnej komórce — starszy bajt. W przykładzie do pary komórek nazwanej PRZEN zostanie wpisana wartość PRZEN-1, czyli adres rozkazu CPL pomniejszony o 1.

Ostatni przykład zasygnalizował, że zamiast pojedynczej stałej lub nazwy można użyć wyrażenia. Budowa wyrażeń jest podobna jak w języku BASIC. Najprostsze asemblery dopuszczają tylko operatory dodawania i odejmowania, jednak większość pozwala korzystać także z mnożenia i dzielenia całkowitego oraz z nawiasów. Korzystając z wyrażeń należy stale mieć na uwadze, że ich wartość jest ostatecznie wyznaczana już w fazie tłumaczenia programu. Wartość wyrażenia jest pojedynczą liczbą, wpisywaną do podanej komórki PAO w chwili ładowania programu do pamięci. Wyrażenie zastosowano też w rozkazie LD B, BETA+5. W naszym przykładzie jest on równoważny rozkazowi: LD B, 55.

Dyrektywa DEFS (dla 8080: DS) rezerwuje podaną liczbę bajtów (np. na tablicę danych) nie określając ich początkowej zawartości. DEFS 30 rezerwuje trzydzieści kolejnych bajtów PAO.

Część asemblerów żąda, aby na końcu programu znajdowała się dyrektywa END. Niektóre asemblery wymagają poprzedzania wszystkich dyrektyw znakiem kropki, np. .ORG, .DEFB, .DEFW. Wbrew pozorom jest to bardzo praktyczne i podnosi czytelność programu.

Wymieniliśmy tylko najważniejsze dyrektywy, akceptowane nawet przez najprostsze asemblery. Na początek zupełnie nam one wystarczą.

3.4. Jak działa asembler?

Jak działa asembler? Jak odbywa się tłumaczenie języka symbolicznego na kod maszynowy? Najlepiej będzie spróbować „ręcznie" wykonać tłumaczenie (asemblację) krótkiego programu, naśladując działanie asemblera. Posłużmy się przykładem z poprzedniego rozdziału. Nie interesuje nas działanie programu, podobnie jak asembler nie analizuje sensu tłumaczonych instrukcji. Zamierzamy tylko przełożyć zapis symboliczny na wiernie odpowiadający mu kod.

Większość asemblerów analizuje tekst programu dwukrotnie (mówimy o pierwszym i drugim przebiegu asemblacji). Właściwe tłumaczenie, którego efektem jest kod maszynowy, odbywa się dopiero w drugim przebiegu. Celem pierwszego przebiegu jest jednoznaczne określenie wartości wszystkich użytych w programie nazw i zbudowanie tzw. słownika nazw (tablicy nazw, tablicy symboli). Po pierwszym przebiegu tablica nazw zawiera przynajmniej wszystkie te nazwy, które zostały w programie zdefiniowane w polu etykiety lub dyrektywą EQU (wszystkie nazwy o określonych wartościach). Oprócz tablicy nazw, tworzonej w trakcie asemblacji, asembler posługuje się kilkoma tablicami stałymi. Najważniejszą spośród nich jest tablica symboli rozkazów, w której zgromadzone są mnemoniczne nazwy rozkazów, ich kody operacji i informacje dodatkowe (długość rozkazu, dozwolone argumenty itp.). W innych tablicach zawarte są nazwy i kody rejestrów wewnętrznych itd. Podczas analizy programu źródłowego asembler porównuje poszczególne słowa programu ze wzorcami zgromadzonymi w tablicach i w ten sposób je rozpoznaje.

Ważnym elementem asemblera jest zmienna, zwana wskaźnikiem rozkazów (symbolicznym licznikiem rozkazów). Zmienna ta zawiera adres, pod jakim zostanie zapisany w PAO najbliższy utworzony bajt kodu. Wartość wskaźnika można zmieniać dyrektywą ORG. Stąd bierze się wymaganie, aby ORG wystąpiła przed pierwszym rozkazem programu w celu zainicjowania (nadania wartości początkowej) wskaźnika rozkazów. W chwili rozpoczęcia każdego przebiegu wskaźnik zawiera wartość standardową (np. O lub adres pierwszej wolnej komórki PAO za obszarem zajętym przez sam asembler).

Asembler „czyta" program źródłowy linia po linii (instrukcja po instrukcji). Najpierw sprawdza, czy na początku wiersza występuje etykieta. Można ją rozpoznać po tym, że kończy ją dwukropek albo (w asemblerach nie wymagających dwukropka) że rozpoczyna się od lewego skraja wiersza. W pierwszym przebiegu po wykryciu etykiety jej nazwa jest umieszczana w tablicy symboli, zaś jako wartość tej nazwy wpisuje się aktualną wartość wskaźnika rozkazów. Jeśli w linii tej wystąpi rozkaz, to wartość nazwy jest równa przyszłemu adresowi tego rozkazu w PAO.

Po sprawdzeniu obecności etykiety i ewentualnym wpisaniu jej do tablicy nazw asembler bada pole operacji. Musi w nim wystąpić nazwa rozkazu lub dyrektywy. Po rozpoznaniu rozkazu, tzn. odnalezieniu rozkazu w tablicy symboli stałych, następuje ustalenie długości jego kodu (ilości bajtów, składających się na rozkaz). Często trzeba w tym celu przeanalizować jeszcze argumenty rozkazu, gdyż rozkaz może mieć kilka wariantów. Wyznaczona długość rozkazu jest dodawana do aktualnej zawartości wskaźnika rozkazów i asembler rozpoczyna analizę następnej linii programu źródłowego. Koniec pierwszego przebiegu kończy się w chwili wykrycia dyrektywy END lub po osiągnięciu końca tekstu programu.

W drugim przebiegu do tablicy symboli nie dopisuje się już nowych wartości (np. etykiety są ignorowane). Więcej uwagi poświęca się rozkazom. Po rozpoznaniu rozkazu analizuje się jego argumenty i w razie potrzeby wylicza ich wartości (np. gdy argument jest wyrażeniem przedstawiającym stałą liczbową lub adres). Tam, gdzie w charakterze argumentu wystąpi nazwa, asembler odszukuje nazwę w tablicy symboli i w miejscu nazwy wstawia jej wartość. Po wyznaczeniu argumentów kompletowany jest rozkaz. Niektóre asemblery wpisują rozkaz od razu w przewidziane miejsce PAO. Inne zapamiętują kody rozkazów w pamięci zewnętrznej. Po zakończeniu asemblacji przetłumaczony program maszynowy musi być wtedy załadowany do PAO z pamięci zewnętrznej.

Rzeczywisty asembler wykonuje jeszcze wiele innych funkcji. Przed wpisaniem nazwy do tablicy sprawdza się np., czy przypadkiem podobna nazwa już nie występuje, a jeśli tak — sygnalizowany jest błąd. Meldunek o błędzie nastąpi też wtedy, gdy w polu operacji wystąpi nazwa nie mająca odpowiednika w tablicy rozkazów, albo gdy rozkaz ma niedozwolone argumenty. Asembler wykrywa też przypadki, gdy argument rozkazu jest nieokreślony (np. skok do nie istniejącej w programie etykiety).

Czas przystąpić do asemblacji. Rozpoczniemy od sporządzania tablicy nazw. Oto jej postać końcowa po pierwszym przebiegu:

Nazwa	Wartość dzies.	Wartość szesn.
BETA	50	32H
START	25600	6400H
PRZEN	25611	640BH
ALFA	25615	640FH
APRZEN	25616	6410H

W drugim przebiegu ignorujemy etykiety i dyrektywy EQU, zajmując się kompletowaniem kodu rozkazów. Tak wygląda rezultat asemblacji: adresy wszystkich rozkazów i zawartości poszczególnych komórek PAO są już ostatecznie określone:

```
   Adres          Kod     ;Przykład programu maszynowego
dzies.  szesn.  szesn.    BETA      EQU   50
                                    ORG   25600
25600    6400    3A       START:    LD    A, (ALFA)
25601    6401    0F
25602    6402    64
25603    6403    06                 LD    B, BETA
25604    6404    32
25605    6405    80                 ADD   A, B
25606    6406    D2                 JP    NC,PRZEN
25607    6407    0B
25608    6408    64
25609    6409    06                 LD    B, BETA+5
25610    640A    37
25611    640B    2F       PRZEN:    CPL
25612    640C    C3                 JP       START
25613    640D    00
25614    640E    64
25615    640F    00       ALFA:     DEFB    0
25616    6410    0A       APRZEN:   DEFW    PRZEN-1
25617    6411    64
```

Kolumna liczb po lewej stronie przedstawia kod programu, jednoznacznie odpowiadający jego symbolicznemu zapisowi po prawej. Ręczna asemblacja jest sensowna tylko w przypadku bardzo krótkich programów, liczących kilka rozkazów. W przyszłości będziemy zajmować się wyłącznie układaniem programów symbolicznych, powierzając ich tłumaczenie asemblerowi. Asembler może dostarczyć podobnego zestawienia jak wyżej, zwanego listingiem asemblacji. Dla większej zwartości pełny kod każdego rozkazu umieszczany jest w jednym wierszu: bajt najmłodszy po lewej, bajt najstarszy po prawej:

```
 1  0000                   ;Przykład programu maszynowego
 2  0000                   BETA     EQU   50
 3  6400                            ORG   25600
 4  6400   3A 0F 64        START:   LD    A, (ALFA)
 5  6403   06 32                    LD    B, BETA
 6  6405   80                       ADD   A, B
 7  6406   D2 0B 64                 JP    NC,PRZEN
 8  6409   06 37                    LD    B, BETA+5
 9  640B   2F              PRZEN:   CPL
10  640C   C3 00 64                 JP       START
11  640F   00              ALFA:    DEFB    0
12  6410   0A 64           APRZEN:  DEFW    PRZEN-1
```

Pierwsza kolumna z lewej przedstawia kolejny numer linii tekstu. Ułatwia to np. odnajdywanie błędnych instrukcji w razie potrzeby wprowadzenia poprawek. Druga kolumna zawiera szesnastkowe adresy poszczególnych rozkazów, trzecia — ich kod.

Do czego służy listing asemblacji? Skoro tłumaczenie wykonuje asembler, znajomość kodów i ich adresów chyba nie powinna być nam potrzebna? Zgoda, lecz tylko pod warunkiem, że program jest bezbłędny! W przeciwnym razie czeka nas żmudne tropienie błędu i analiza pracy programu krok po kroku. Wtedy znajomość lokalizacji poszczególnych rozkazów i danych będzie wprost nieodzowna.

4. POZNAJEMY ROZKAZY, BUDUJEMY PROGRAMY

Znajomość zasad notacji nie wystarcza do samodzielnego konstruowania programów. Trzeba jeszcze poznać dostępny budulec, czyli rozkazy procesora. Zaczniemy od najbardziej elementarnych i najczęściej spotykanych. Będziemy się przy tym interesować wyłącznie działaniem rozkazu, możliwością jego wykorzystania w programie oraz sposobem zapisu w języku symbolicznym, nie zaś sposobem zestawiania odpowiadającego mu kodu maszynowego. Tym niechaj zajmie się asembler.

4.1. Podstawowe narzędzie: rozkazy przesłań

Z punktu widzenia danych język maszynowy jest mniej „demokratyczny" od języków wyższego poziomu. Zamiast równoprawnych zmiennych mamy podział na komórki pamięci i rejestry wewnętrzne procesora o różnym stopniu uprzywilejowania. Aby wykonać na danych określone operacje, należy wpierw wprowadzić je do odpowiednich rejestrów. Z tego powodu najczęstszym zadaniem procesora jest przesyłanie danych między rejestrami a pamięcią oraz między samymi rejestrami. Grupa rozkazów jest szczególnie liczna. Wszystkie one mają wspólną nazwę mnemoniczną LD (ang. LOAD — ładuj).

Najprostsze są przesłania międzyrejestrowe. Pojedynczym rozkazem można przepisać bajt z dowolnego rejestru źródłowego do dowolnego rejestru przeznaczenia. Jako pierwszy wymieniamy zawsze rejestr przeznaczenia (ten, którego bajt jest wpisywany), a dopiero potem — źródło informacji:

```
LD   A, B    ;skopiuj bajt z rejestru B do A
LD   B, L    ;skopiuj bajt z rejestru L do B
```

Lista rozkazów w Dodatku B zawiera takie pozycje jak LD A, A; LD C, C itd. Rozkazy takie niczego nie zmieniają i w zasadzie są całkowicie zbędne. Czym wytłumaczyć ich istnienie? Analizując kod operacji przesłań międzyrejestrowych, zauważylibyśmy, że dwa najstarsze bity są niezmienne — 01 — zaś pozostałe dzielą się na dwie grupy trzybitowe. Każdy rejestr jest zakodowany w postaci kombinacji trzech bitów: B — 000, C — 001, A — 111, itd. Trzy starsze bity przedstawiają rejestr docelowy, trzy młodsze — źródłowy. Kod: 01111000B odpowiada więc rozkazowi LD A, B. Procesor nie bada, czy kod rejestru w obu grupach nie jest przypadkiem identyczny, lecz posłusznie wykonuje bezsensowny rozkaz. Widzimy zatem, że procesor jest w gruncie rzeczy urządzeniem prymitywnym i niedoskonałym. Cała „inteligencja" komputera tkwi w jego oprogramowaniu.

Gdy zajdzie potrzeba przepisania jednej pary rejestrów do drugiej, np. HL do BC, trze-

ba oddzielnie przepisać obydwa bajty. Kolejność jest dowolna, ale trzeba zachować odpowiedniość bajtów starszych i młodszych:

```
LD  B, H
LD  C, L
```

Kiedy trzeba zamienić zawartość rejestrów, np. D i B, nie obejdzie się bez pomocy trzeciego rejestru, np. A:

```
LD  A, D
LD  D, B
LD  B, A
```

Do zamiany pary rejestrów trzeba aż 6 rozkazów. Wkrótce przekonamy się, że para HL jest bardzo uprzywilejowana i konieczność wymiany jej zawartości z pozostałymi parami rejestrów zachodzi szczególnie często. Upraszczają ją specjalne instrukcje wymiany EX (ang. exchange — wymiana):

```
          EX  HL, DE
oraz      EX  HL, BC
```

Pierwsza zamienia zawartości par HL i DE, druga — HL i BC.

Częstym zadaniem jest wpisywanie do rejestrów stałych liczbowych, do czego służą specjalne rozkazy LD. Pierwszym argumentem jest rejestr, drugim — jednobajtowa stała. Stałą można podać w sposób dogodny dla programisty:

```
LD  A, 111   ;do akumulatora wpisz stałą 111
LD  H, 0FFH  ;w rejestrze H umieść stałą 0FFH
             ;(dziesietnie: 255 lub -1)
```

Chcąc załadować do pary rejestrów, np. HL, szesnastobitową stałą, np. 16387, można uczynić to dwoma rozkazami:

```
LD  L, 3        ;L -rejestr młodszy, H -starszy
LD  H, 64       ;256*H + L = 256*64+3 = 16387
```

Ten sam skutek osiągniemy w prostszy sposób specjalnymi instrukcjami ładowania par rejestrów:

```
LD  BC, 0       ;wyzeruj rejestry B i C
LD  DE, 0FFFFH  ;wpisz do DE stałą 65535
LD  HL, 16387   ;wpisz do HL stałą 16387
```

Zamiast dwóch rozkazów i czterech bajtów potrzeba jednego rozkazu i trzech bajtów. Co ważniejsze, program tłumaczący sam dzieli liczbę na bajt młodszy i starszy, uwalniając nas od tej niewdzięcznej czynności.

Chcąc odczytać lub zapisać wskazaną wprost komórkę PAO, trzeba posłużyć się akumulatorem. Dla innych rejestrów nie przewidziano odpowiednich rozkazów:

```
LD  A, (32769)   ;wpisz do A bajt z kom. 32769
LD  (32770), A   ;skopiuj  A do komórki  32770
```

Ujęcie argumentu w nawiasy sygnalizuje, że to nie on sam uczestniczy w operacji, lecz wskazywana przezeń komórka pamięci, czyli że argument jest adresem. Jest to konwencja uniwersalna, obowiązująca także w innych rozkazach.

Pytanie 1. Czym różnią się rozkazy: LD A, 45 i LD A, (45)? Rozkaz: LD (250), A jest legalny. Czy mógłby istnieć rozkaz: LD 250, A?

Przy operacjach na ciągach bajtów podawanie adresu każdego z nich wprost w rozkazie byłoby niewygodne i najczęściej niemożliwe. Korzystamy wtedy z tzw. adresacji pośredniej. Adres komórki PAO jest zawarty we wskazanej parze rejestrów:

```
LD  A, (BC)    ;wpisz do A bajt o adresie w BC
LD  A, (DE)    ;wpisz do A bajt o adresie w DE
LD  (BC), A    ;kopiuj A do PAO pod adres z BC
LD  (DE), A    ;kopiuj A do PAO pod adres z DE
```

Nazwa pary rejestrów występuje w nawiasach, gdyż para zawiera nie operand, lecz tylko jego adres. Adres podawany jest więc pośrednio. Pary BC i DE mogą służyć do adresacji tylko przy wymianie danych między PAO i akumulatorem. Użycie w adresacji pośredniej pary rejestrów HL pozwala przesyłać dane między PAO a dowolnym rejestrem:

```
LD  A, (HL)    ;wpisz do A bajt  o  adresie w HL
LD  (HL), C    ;kopiuj C do PAO  pod  adres z HL
LD  B, (HL)    ;kopiuj do B bajt spod adresu w HL
```

Komórka PAO, zaadresowana zawartością pary HL może wystąpić w większości rozkazów zamiast rejestru wewnętrznego. Przykładem niech będzie umieszczanie stałej w komórce PAO:

```
LD  HL,   50000      |     LD  A,  100
LD  (HL), 100        |     LD  (50000), A
```

Obie pary rozkazów umieszczają liczbę 100 w komórce PAO nr 50000. Ostatnie dwa rozkazy pozwalają przesyłać między parą HL a PAO dwubajtowe słowa:

```
LD  HL, (60000)   ;wpisz do HL słowo spod  adr. 60000
LD  (60000), HL   ;zaw. HL wpisz w PAO pod adr. 60000
```

Pierwszy rozkaz umieści w L bajt z komórki 60000, a w H — bajt spod adresu 60001. Drugi rozkaz skopiuje L do komórki 60000, H do 60001. Utrzymana jest zasada, że młodszy bajt zajmuje adres niższy.

Pytanie 2. Jak załadować do HL słowo spod adresu 20000, nie posługując się rozkazem: LD HL, (20000)?

Wszystkie rozkazy LD i EX przesyłają bajty lub słowa w postaci niezmienionej, nie wpływając na stan żadnego z bitów stanu (warunków).

4.2. Najprostsze operacje arytmetyczne i logiczne

Przesyłanie danych z jednego miejsca w inne w procesie „obróbki" informacji pełni zwykle funkcję tylko usługową. Typowymi operacjami „przetwórczymi" są: dodawanie i odejmowanie.

Chcąc dodać dwie liczby jednobajtowe, należy jedną z nich umieścić w akumulatorze, a drugą w dowolnym rejestrze. Przy odejmowaniu do akumulatora wpisujemy odjemną, zaś odjemnik do innego rejestru. Wynik znajdzie się w akumulatorze. Będzie to regułą we wszystkich dwuargumentowych operacjach na liczbach jednobajtowych. Dodamy zawartości rejestrów B i D, od sumy odejmiemy liczbę z rejestru E:

```
LD  A, B    ;zawartość  B wpisz do akumulatora
ADD A, D    ;dodaj  do akumulatora liczbę  z D
SUB E       ;od  treści  akumulatora odejmij E
```

Zapis ADD i SUB jest trochę niekonsekwentny, prawda? W pierwszym przypadku wymieniamy oba operandy, w drugim tylko jeden. Przy dodawaniu liczb jednobajtowych pierwszym argumentem rozkazu ADD musi być zawsze rejestr A. Drugi argument może być dowolnym rejestrem albo komórką pamięci, wskazaną za pośrednictwem pary HL:

```
ADD  A, (HL)   ;dodaj  do  A liczbę z PAO
SUB  (HL)      ;odejmij od A liczbę z PAO
```

Jeśli drugi operand jest stałą liczbową, nie trzeba umieszczać go w rejestrze. Wolno skorzystać z wariantów rozkazów ADD i SUB z tzw. adresacją bezpośrednią (stała jest podana jako drugi bajt rozkazu):

```
ADD  A, 35    ;dodaj do akumulatora stałą 35
SUB  42H      ;odejmij od A stałą 66
```

ADD i SUB wpływają na wszystkie bity stanu zależnie od wyniku operacji. Bit C jest ustawiany w zależności od wartości sumy lub różnicy operandów, rozumianych jako liczby bez znaku. CY=1, gdy przy dodawaniu suma = 255, zaś przy odejmowaniu różnica = 0. Bit CY nie ma jednak nic wspólnego z bitem S, traktującym wynik jako liczbę ze znakiem.

Pytanie 1. Do czego można wykorzystać rozkaz ADD A, A?

Odmianą SUB jest rozkaz CMP. Jedyna różnica tkwi w fakcie, że CMP wykonuje odejmowanie „na niby", nie zmieniając zawartości akumulatora, lecz ustawiając bity stanu tak, jak uczyniłby to rozkaz SUB. Rozkaz CMP jest bardzo przydatny przy porównywaniu ze sobą dwóch liczb:

```
CMP  B     ;porównaj zawartości rejestrów A i B
CMP  (HL)  ;porównaj akumulator  z komórką  PAO
CMP  65    ;porównaj  akumulator  ze stałą   65
```

Sprawa bitu Z jest oczywista: przyjmie wartość 1 tylko wtedy, gdy obie porównywane liczby są równe. Bity: CY i S wskazują, która z dwu liczb jest większa. Bitu CY używamy, gdy interpretujemy obie liczby bez znaku (jako nieujemne, od 0 do 255), bitu S — ze znakiem

(–128:+127). Bity te są ustawione, gdy porównywana liczba jest większa od zawartości akumulatora, i skasowane, gdy mniejsza lub równa. Może się zdarzyć, że bit CY będzie wyzerowany, a S — ustawiony, i odwrotnie:

```
LD  A, 5     ;załaduj  do  akumulatora  liczbą  5
CMP 255      ;porównaj zaw. A z 255  (inaczej:-1)
```

Rozkaz „nie wie", jak interpretujemy wartości, i ustawia bity CY i S oddzielnie, na wypadek obu wariantów. W powyższym przykładzie CY=1 (255 jest większe od 5), zaś S=0 (–1 mniejsze od +5).

Przy odliczaniu itd. często zachodzi konieczność zmniejszania lub zwiększania zawartości rejestrów o 1 (inkrementacji i dekrementacji). Realizują to specjalne rozkazy INC i DEC. Argumentem może być dowolny rejestr, lub komórka PAO, wskazana za pośrednictwem pary HL:

```
INC   A     ;zwiększ o 1 zawartość akumulatora
DEC   B     ;zmniejsz o 1 zawartość rejestru B
INC  (HL)   ;zwiększ o 1 zawartość komórki PAO
```

Gdy rejestr lub komórka PAO zawiera 255, kolejna inkrementacja zeruje ją. Dekrementacja rejestru zawierającego 0 daje 255 (inaczej: –1).

Po inkrementacji lub dekrementacji argumentów jednobajtowych bity stanu: Z i N odpowiadają wynikowi, a bit C zachowuje pierwotną wartość.

Inkrementację lub dekrementację par rejestrów umożliwiają specjalne rozkazy:

```
INC  BC   ;zwiększ  o 1 zawartość BC
INC  DE   ;zwiększ  o 1 zawartość DE
INC  HL   ;zwiększ  o 1 zawartość HL
DEC  BC   ;zmniejsz o 1 zawartość BC
DEC  DE   ;zmniejsz o 1 zawartość DE
DEC  HL   ;zmniejsz o 1 zawartość HL
```

Niech HL zawiera początkowo 0. Czym różni się rozkaz INC L od INC HL? Początkowo niczym istotnym. Gdy jednak zawartość L osiągnie stan 255, to po kolejnym rozkazie INC L rejestr L zostanie po prostu wyzerowany, bez dalszych skutków. INC HL traktuje rejestry H i L jako całość, a zatem wyzeruje L, zwiększając o 1 zawartość H (uwzględni przeniesienie).

Rozkazy inkrementacji i dekrementacji par rejestrów nie wpływają na bity stanu. Niebawem przekonamy się, że ten pozorny paradoks dowodzi dalekowzroczności projektantów procesora!

Przejdźmy do operacji logicznych. Negacją logiczną nazywamy zmianę wartości poszczególnych bitów. Jeśli przed negacją bajt=01110001B, to po negacji: 10001110B. Negację akumulatora wykonuje rozkaz:

```
CPL       ;neguj logicznie akumulator
```

CPL nie wpływa na bity stanu.

Pytanie 4. Jak zmienić na przeciwny znak liczby zawartej w akumulatorze (zanegować ją arytmetycznie)?

Suma logiczna i iloczyn logiczny to jedne z najczęstszych operacji, realizowane przez

rozkazy OR i AND. Pierwszy operand i wynik operacji znajdują się w akumulatorze, zaś drugi operand może być zawartością rejestru, komórki PAO zaadresowanej przez parę HL lub stałą, podaną w rozkazie (jak w ADD i SUB):

```
OR  B      ;oblicz sumę logiczną A i B
AND L      ;wyznacz iloczyn logiczny A i L
OR  (HL)   ;oblicz sumę logiczną A i komórki PAO
AND 15     ;iloczyn log. rejestru  A i liczby 15
```

Wszystkie operacje logiczne zawsze kasują bit CY i zmieniają wartości bitów Z i S zgodnie z wynikiem rezultatu. Rozkaz OR pozwala ustawiać, zaś AND — zerować pojedyncze bity lub ich grupy. W poniższym przykładzie zerowane są cztery najstarsze bity akumulatora i ustawiany bit najstarszy — niezależnie od ich pierwotnej wartości:

```
AND 15   ;15 = 00001111B, zeruj bity nr 0-3
OR  128  ;128= 10000000B, ustaw  bit nr 7
```

Rozkaz XOR wyznacza różnicę symetryczną akumulatora ze wskazanym bajtem. Dopuszczalne argumenty — jak dla OR lub AND. Po operacji wyzerowane są te bity akumulatora, którym odpowiadały bity operandów o zgodnych wartościach, np. 0 i 0 lub 1 i 1. Na tych pozycjach, gdzie bity były różne, w wyniku będzie 1. Rozkaz XOR może służyć do negacji logicznej (zmiany wartości na przeciwną) pojedynczych bitów. Chcemy oto zanegować bit nr 2 (trzeci od prawej) w akumulatorze:

```
XOR  4   ;różnica symetryczna A ze stałą 00000100B
```

Pytanie 5. Często trzeba zerować akumulator. Nie ma do tego celu specjalnego rozkazu. Jak wyzerować akumulator, nie używając rozkazu LD?

4.3. Skoki warunkowe i bezwarunkowe

Programowanie pętli i rozgałęzień jest możliwe dzięki rozkazom skoków. Argumentem rozkazu skoku jest adres rozpoczynający kolejną sekwencję rozkazów, gdyż dotychczasowa została przerwana. Najprostszy jest skok bezwarunkowy, który wykona się zawsze, bez względu na bity stanu:

```
JP  32769   ;skocz bezwarunkowo do  adresu  32769
JP  PETLA   ;skocz bezwarunkowo do etykiety PETLA
```

W rozkazach skoku warunkowego skok dojdzie do skutku tylko pod warunkiem, że wybrany bit stanu posiada określoną wartość. Dla każdego bitu stanu mamy zatem parę skoków warunkowych. Oto skoki warunkowe odnoszące się do interesującej nas trójki bitów stanu:

```
JP NZ, ADRES1   ;skocz, jeśli Z= 0 (nie zero)
JP Z,  ADRES2   ;skocz, jeśli Z= 1 (zero)
JP NC, ADRES3   ;skocz, jeśli CY=0 (nie carry)
JP C,  ADRES4   ;skocz, jeśli CY=1 (carry)
JP P,  ADRES5   ;skocz, jeśli S= 0 (plus)
JP M,  ADRES6   ;skocz, jeśli S= 1 (minus)
```

Jeśli warunek nie jest spełniony, skok nie nastąpi, a jako następny wykona się rozkaz położony w pamięci bezpośrednio za rozkazem skoku. Skoki umożliwiają warunkowe wykonywanie grup instrukcji. Trzeba zbadać zawartość komórki PAO nr 17000 i, jeśli jest ona równa 32, zwiększyć o 1 zawartość rejestru B:

```
        LD   A, (17000)  ;ładuj do A zaw. kom. 17000
        CP   32          ;porownaj  A  z  liczbą 32
        JP   NZ, NIEROW  ;jeśli nierówne,  przeskocz
        INC  B           ;zwiększ  o  1  zawartość B
NIEROW: LD   A, B        ;......dalszy ciąg programu
```

Gdy komórka PAO zawierała inną liczbę niż 32, rozkaz INC B zostałby po prostu „przeskoczony".

Pytanie 6. Jeśli zawartość komórki 30000 jest zerowa, należy skoczyć pod adres ZER. Jak to zapisać?

Ważnym zastosowaniem rozkazów skoku jest organizacja pętli programowych. Poniższy program wyznacza resztę z dzielenia akumulatora przez 10 (zaw. A — liczba ze znakiem). Stała 10 jest odejmowana od A tak długo, aż zawartość A stanie się ujemna. Wtedy wystarczy „unieważnić" ostatnie odejmowanie, ponownie dodając 10. Zawartość akumulatora stanie się wtedy resztą z dzielenia jego pierwotnej zawartości przez 10:

```
PETLA: SUB 10        ;odejmij stałą 10 od akumulatora
       JP  P, PETLA  ;jeśli wynik>=0, powtórz odejmow.
       ADD 10        ;odtwórz A sprzed ostat. odejmow.
```

Oto inna pętla, mnożąca zawartość rejestrów B i C. Zastosujemy najprymitywniejszy sposób mnożenia: będziemy tyle razy dodawać do akumulatora liczbę z C, ile wynosi wartość B. Na początku niech akumulator zawiera 0; po zakończeniu pętli będzie zawierał iloczyn B i C:

```
MNOZ:  LD   A, 0     ;wyzeruj zawartość  akumulatora
POWT:  ADD  A, C     ;dodaj do akumulatora wartość C
       DEC  B        ;zmniejsz o 1 zawartość rej.  B
       JP   NZ, POWT ;jeśli nie 0, powtórz dodawanie
```

Początkowa zawartość rejestru B określa ilość powtórzeń pętli. Po każdym powtórzeniu zawartość B jest zmniejszana o 1 (DEC B), równocześnie ustawiane są bity stanu, m.in. Z. Gdy po kolejnym powtórzeniu wartość B zostanie wyzerowana (bit Z=1), rozkaz skoku nie wykona się i pętla zostanie zakończona. Rejestr B odegrał rolę licznika powtórzeń.

Pytanie 7.

```
LD A, (30000)
ADD 20
JP P, DODATN
JP Z, ZERO
```

Gdy wynik dodawania=0, powinien nastąpić skok pod adres ZERO, zaś gdy wynik >0 skok pod adres DODATN. Wynik interpretujemy jako liczbę ze znakiem. Program działa błędnie. Dlaczego?

Poznajemy jeszcze jeden skok — tzw. bezwarunkowy skok pośredni. Program przejdzie bezwarunkowo do adresu, podanego jako zawartość pary HL w chwili wykonywania skoku:

```
JP (HL)     ;skocz do adresu zawartego w HL ;
```

Żaden z rozkazów skoku nie zmienia wartości bitów stanu ani rejestrów procesora (poza licznikiem programu PC).

4.4. Analizujemy programy

Zapoznawszy się z podstawowymi rozkazami spróbujmy swych sił analizując kilka gotowych programów. Przy okazji poznamy metody rozwiązywania typowych problemów, występujących przy programowaniu w języku asemblera. Postanowiliśmy nie zajmować się kodami rozkazów. Poniżej przedstawiono jednak wyjątkowo obraz programu w pamięci PAO. Niech posłuży to za przypomnienie i ilustrację bezpośredniego związku między programem w języku symbolicznym, a kodem maszynowym w PAO.

Częstą operacją jest wypełnienie jakiegoś obszaru PAO stałą zawartością, np. bajtami o wartości 0. Poniższy program zeruje blok pamięci o długości 100 bajtów, zaczynający się od adresu 16384:

```
F000                  ORG  0F000H
F000 21 00 40 ZERUJ:  LD   HL,    16384  ;HL= adres pocz.
F003 06 64            LD   B,     100    ;B= ilość bajtow
F005 36 00    PETLA:  LD   (HL),  0      ;wpisz 0  do PAO
F007 23               INC  HL            ;dodaj 1  do  HL
F008 05               DEC  B             ;odejmij 1  od B
F009 C2 05 F0         JP   NZ,    PETLA  ;skocz,gdy nie 0
```

Program zawiera pętlę, wykonującą się 100 razy. Parze HL przypadnie rola wskaźnika kolejnych komórek zerowanego obszaru pamięci. Na początek wpisujemy do HL adres pierwszej komórki bloku PAO LD HL, 16384. Rejestr B spełni w naszym programie rolę licznika powtórzeń pętli. Wpisujemy do niego 100 LD B, 100. Poczyniliśmy niezbędne przygotowania, czas przystąpić do sedna sprawy. Wpisujemy liczbę 0 do komórki, której adres zawarty jest w parze HL LD (HL), 01. Para HL zawiera 16384 i komórka o tym numerze zostanie wyzerowana. Następną czynnością jest zwiększenie o 1 liczby w parze HL INC HL. Od tej chwili HL zawiera już 16385. Rozkaz DEC B wykonuje dwie operacje: odejmuje od zawartości rejestru B stałą 1 i w zależności od wyniku odejmowania ustawia bity warunków, w szczególności bit Z (wskaźnik zero). Po pierwszej dekrementacji B zawiera 99, rezultat NIE JEST ZEREM, a więc bit Z jest ZEROWANY?

Rozkaz JP NZ, PETLA jest skokiem warunkowym. Skok do adresu PETLA zostanie wykonany, jeśli bit Z jest WYZEROWANY (NZ=NIEPRAWDA, ŻE ZERO). Za pierwszym razem skok oczywiście nastąpi. Do licznika rozkazów PC zostanie wpisany adres, zamarkowany etykietą PETLA, czyli F005. Teraz czwórka instrukcji będzie powtarzana cyklicznie. Przy każdym powtórzeniu para HL ulegnie zwiększeniu o 1 i wskaże adres następnej komórki kasowanego bloku PAO (16385, 16386, 16387,...,16483). Odwrotnie rejestr B: jego zawartość stopniowo się zmniejsza (99,98,97,...,1). Po kolejnych dekrementacjach zawartość B jest ciągle większa od 0, bit Z za każdym razem jest zerowany. Wreszcie B zawiera 1 i po raz setny wykonywany jest rozkaz DEC B. Po odjęciu 1 w B pozostaje 0, wskaźnik Z jest ustawiany, skok warunkowy nie wykonuje się i program przechodzi do następnych rozkazów.

Dla lepszej ilustracji rzućmy okiem na program w języku BASIC, wykonujący te same czynności:

```
10 LET  HL= 16384
20 LET  B = 100
30 POKE HL, 0
40 LET  HL= HL+1
50 LET  B = B -1
60 IF B<>0 THEN GOTO 30
```

Analogia jest dość odległa: zmienne HL i B nie mają oczywiście nic wspólnego z rejestrami procesora o tej samej nazwie, a ich wartości są przechowywane w wybranych przez interpreter języka BASIC komórkach PAO.

Inna często spotykana czynność to przesyłanie (kopiowanie) jednego fragmentu pamięci w inny. Chcemy spowodować szybkie pojawienie się na ekranie złożonego rysunku. Co robimy? Przygotowujemy rysunek. Po czym kopiujemy zawartość pamięci ekranu do bezpiecznego obszaru pamięci. Gdy rysunek powinien szybko się ukazać (np. mapa terenu w grze), przepisujemy zapamiętane dane z powrotem do pamięci ekranu. W języku maszynowym potrwa to ułamek sekundy.

Poniższy program kopiuje 1024 bajty począwszy od komórki 16384 pod adres 32768:

```
F00C                    ORG  0F00CH
F00C 21 00 40 KOPIA:    LD   HL,    16384   ;HL= adres oryginału
F00F 11 00 80           LD   DE,    32768   ;DE= adres kopii
F012 01 00 04           LD   BC,    1024    ;BC= ilość komórek
F015 7E       PETLA:    LD   A,     (HL)    ;odczytaj bajt do A
F016 12                 LD   (DE),  A       ;wyślij bajt do PAO
F017 23                 INC  HL             ;zwiększ o 1 zaw. HL
F018 13                 INC  DE             ;zwiększ o 1 zaw. DE
F019 0B                 DEC  BC             ;odejmij 1 od BC
f01A 78                 LD   A,     B       ;przepisz B do A
F01B B1                 OR   C              ;suma logiczna A z C
F01C C2 15 F0           JP   NZ,    PETLA   ;skocz, jeżeli nie 0
```

Tym razem niezbędne okazało się zaangażowanie wszystkich rejestrów. Para HL posłuży za wskaźnik kolejnych komórek „oryginału" (źródła), czyli tego fragmentu PAO, który ma zostać skopiowany. Para DE będzie wskaźnikiem komórek obszaru kopii (celu). Na początku wpisujemy do HL i DE adresy początkowe (adresy pierwszej komórki) obydwu obszarów.

Operację przepisywania zawartości poszczególnych komórek trzeba będzie powtórzyć 100 razy. Pojedynczy rejestr nie pomieści tak dużej liczby (maks. 255), w charakterze licznika użyjemy zatem pary rejestrów BC. Kopiowanie może się rozpocząć. Przepisywanie bajtu odbywa się „na dwa pas". Pierwszy krok to odczytanie bajtu z komórki PAO, wskazanej przez parę HL (LD A,/HL/), i wpisanie go do akumulatora. Krok drugi polega na zapisaniu zawartości A pod adres z pary DE. Pierwszy bajt skopiowany, trzeba więc „przestawić" wskaźniki na następny. Wykonują to rozkazy INC HL i INC DE. Dzięki równoczesnej inkrementacji par HL i DE różnica ich zawartości pozostaje niezmieniona.

Pozostało zarejestrować powtórzenie pętli, zmniejszając o 1 licznik BC rozkazem DEC BC. Tutaj pojawia się problem typowy dla procesorów 8080/Z80. W odróżnieniu od rozkazów dekrementujących pojedyncze rejestry (np. DEC B), dekrementacja par rejestrów nie wpływa na bity warunków. Inaczej mówiąc, po wykonaniu rozkazu DEC BC wskaźnik zera Z nie informuje nas, czy para BC zawiera liczbę 0. Trzeba uciec się do wybiegu.

Jak stwierdzić, że zawartość pary BC jest zerem? Wystarczy sprawdzić, czy wszystkie

bity rejestrów B i C są wyzerowane. Najłatwiej posłużyć się sumą logiczną. Kopiujemy rejestr B do akumulatora [LD A,B], a następnie wyznaczamy sumę logiczną akumulatora i rejestru C [OR C]. Odpowiada ona sumie logicznej B i C. Jeśli choć jeden bit w którymkolwiek z rejestrów będzie ustawiony, suma logiczna okaże się niezerowa. Rozkaz OR C ustawia bity warunków, a więc następujący po nim rozkaz skoku warunkowego JP NZ, PETLA wykona się zgodnie z naszymi intencjami. Pętla będzie powtarzana dopóty, dopóki zawartość pary BC nie zostanie wyzerowana, co nastąpi dopiero po 1020 powtórzeniach.

Pytanie 8. Czy zamiast rozkazu sumy logicznej OR C wolno było użyć rozkazu sumy arytmetycznej ADD A, C?

Zaproponowany sposób kopiowania nie jest jedynym dozwolonym. Równie dobrze można zacząć kopiować od ostatniej komórki bloku PAO. Oto wariant programu kopiującego, działającego „wspak":

```
F00C                   ORG  0F00CH
F00C 21 FF 43 KOPIA:  LD   HL,    17407   ;HL= 16384+1023
F00F 11 FF 83         LD   DE,    33791   ;DE= 32768+1023
F012 01 00 04         LD   BC,    1024    ;BC= ilość komórek
F015 7E       PETLA:  LD   A,     (HL)    ;odczytaj bajt  do A
F016 12               LD   (DE),  A       ;wyślij bajt do  PAO
F017 2B               DEC  HL             ;zmniejsz o 1 zaw.HL
F018 1B               DEC  DE             ;zmniejsz o 1 zaw.DE
F019 0B               DEC  BC             ;odejmij 1 od BC
F01A 78               LD   A,     B       ;przepisz B do A
F01B B1               OR   C              ;suma logiczna A z C
F01C C2 15 F0         JP   NZ,    PETLA   ;skocz, jeżeli nie 0
```

Inkrementację par HL i DC zastąpiła dekrementacja. Obydwie pary muszą początkowo zawierać adres ostatniej komórki obszarów: źródłowego i docelowego. Zauważmy, że adres ostatniej komórki obszaru liczącego 1024 bajty różni się od adresu początkowego nie o 1024, lecz o 1023!

Pytanie 9. W powyższym przykładzie wynik kopiowania nie zależał od kierunku (od tego, czy posuwaliśmy się od adresów niższych do wyższych czy odwrotnie). Czy może zajść sytuacja, w której poprawność kopiownia będzie uzależniona od kierunku?

4.5. „Prawdziwa" arytmetyka

Co począć, gdy zajdzie potrzeba obliczenia sumy lub różnicy liczb wielobajtowych (np. szesnastobitowych)? Rozkazy ADD i SUB nie wystarczą, gdyż operują tylko na pojedynczych bajtach.

Dodając lub odejmując liczby wielobajtowe trzeba zawsze zaczynać od bajtów najmłodszych, a przy wszystkich bajtach starszych uwzględniać przeniesienie (carry). Służą do tego rozkazy ADC (dodaj z przeniesieniem) i SBC (odejmij z przeniesieniem). Jeśli przed wyko-

naniem rozkazu ADC bit C miał wartość 1, wynik dodawania zostanie powiększony o 1. Jeśli przed wykonaniem SBC bit CY=1, wynik odejmowania będzie dekrementowany (w przypadku odejmowania bit CY przedstawia pożyczkę).

Zamierzamy obliczyć różnicę liczb dwubajtowych. Odjemna znajduje się w parze HL, odjemnik — w DE. Wynik ma być umieszczony w HL zamiast odjemnej:

```
ODEJM2: LD  A, L  ;młodszy bajt odjemnej do A
        SUB E     ;odejmij młodszy bajt odjemnika
        LD  L, A  ;wyślij młodszy bajt wyniku do L
        LD  A, H  ;starszy bajt odjemnej do A
        SBC D     ;odejmij starszy bajt odjemnika
        LD  H, A  ;wpisz starszy bajt wyniku do H
```

Niech HL zawiera 519 (bajt starszy= 2, młodszy= 7), zaś DE — 510 (starszy= 1, młodszy= 254). Odejmujemy młodsze bajty: 7−254=−247. Nastąpi pożyczka w wysokości 256 od starszego bajtu. W akumulatorze pozostanie 256−247=9, zaś bit zostanie ustawiony, sygnalizując pożyczkę. Wartość 9 zostaje wpisana do L, w akumulatorze odbywa się odejmowanie starszych bajtów: 2−1= 1, ale ponieważ CY=1, rozkaz SBC zmniejszy wynik o 1, zatem rezultat brzmi: 0. Ostatecznie para HL zawiera 256*0+9=9, czyli 519−510. Zgadza się? Gdyby zamiast SBC wystąpił rozkaz SUB, odejmując starsze bajty bez uwzględnienia pożyczki otrzymalibyśmy: 256*1+9=265, czyli błąd!

Pytanie 10. Jak zmienić na przeciwny znak zawartości pary HL (zanegować arytmetycznie HL)?

Należy dodać dwie dwubajtowe liczby, umieszczone w PAO pod adresami: 20000 i 20002. Wynik należy umieścić także w PAO, pod adresem 21000. Posłużmy się adresacją pośrednią. Adres pierwszego składnika wpiszemy do BC, drugiego — do HL, adres przyszłej sumy — do DE:

```
DODAJ2: LD  BC, 20000  ;ładuj do BC, DE i HL adresy
        LD  HL, 20002  ;młodszych bajtów obydwu
        LD  DE, 21000  ;składników i przyszłej sumy
        LD  A,  (BC)   ;dodaj w rejestrze A młodsze
        ADD (HL)       ;bajty składników i zapisz
        LD  (DE), A    ;w pamięci młodszy bajt sumy
        INC BC         ;zwiększ o 1 pary BC,HL i DE.
        INC HL         ;ustawiając adresy starszych
        INC DE         ;bajtów składników i sumy
        LD  A,  (BC)   ;dodaj starsze bajty
        ADC A,  (HL)   ;uwzględniając przeniesienie
        LD  (DE), A    ;i zapisz w PAO starszy bajt
```

Dodawanie liczb dwubajtowych w parach rejestrów ułatwiają specjalne rozkazy:

```
ADD HL, BC  ;do zawartości HL dodaj BC, wynik w HL
ADD HL, DE  ;do zawartości HL dodaj DE, wynik w HL
ADD HL, HL  ;wyznacz w HL sumę HL+HL (pomnóż HL*2)
```

Rejestr HL spełnia tu funkcję jak gdyby „szesnastobitowego akumulatora". Uwaga! W odróżnieniu od rozkazu ADD A,..., rozkaz ADD HL,... wpływa tylko na bit przeniesienia CY.

Pozostałe bity stanu nie zmieniają swej wartości! Trzeba powiększyć zawartość rejestru HL o 4096:

```
LD  BC, 4096  ;ładuj drugi składnik do BC
ADD HL, BC    ;wyznacz sumę HL+BC  i wpisz ją do HL
```

Pytanie 11. Jak najprościej pomnożyć zawartość pary HL przez stałą 17? Wolno użyć rejestru BC.

Procesor „nie umie" mnożyć i dzielić. Mnożenie najprościej zastąpić wielokrotnym dodawaniem. Niech czynniki znajdują się w parach: BC i DE, zaś iloczyn umieścimy w HL. Czynniki mogą być dowolnymi liczbami z zakresu 1—65535, byle tylko ich iloczyn nie przekraczał wartości 65535:

```
MNOZ2: LD  HL, 0    ;początkowa  wartość  sumy = 0
MNP:   ADD HL, DE   ;dodaj do sumy pierwszy czynnik
       DEC BC       ;zmniejsz o 1  wartość licznika
       LD  A,  C    ;oblicz w  A  sumę logiczną obu
       OR  B        ;bajtów  licznika, jeśli  nie 0
       JP  NZ, MNP  ;to  BC>0  i  powtórz dodawanie
```

BC spełnia funkcję licznika pętli. Para DE jest dodawana tyle razy do ML, ile wynosiła pierwotna wartość liczby w BC.

Pytanie 12. Program MNOZ2 pracuje błędnie, gdy czynnik BC = 0. Dlaczego? Jak temu zaradzić?

Należy podzielić HL przez DE. Co znaczy „iloraz HL i DE"? Po prostu: „ile razy DE mieści się w HL". Przyjmijmy, że dzielna i dzielnik są liczbami dodatnimi, bez znaku. Będziemy tak długo odejmować DE od HL, aż przy odejmowaniu STARSZYCH bajtów wystąpi pożyczka. Będzie to znaczyło, że zawartość HL stała się mniejsza, niż DE. Każde odejmowanie będziemy rejestrować:

```
DZIEL2: LD  BC, 0    ;początkowa wartość licznika =0
DZP:    INC BC       ;rejestruj kolejne  powtórzenie
        LD  A,  L    ;odejmij liczbę zawartą w DE od
        SUB E        ;liczby znajdującej  się  w  HL
        LD  L,  A    ;jeśli wartość odjemnika będzie
        LD  A,  H    ;większa  od wartości odjemnej,
        SBC D        ;przy   odejmowaniu   starszych
        LD  H,  A    ;bajtów wystąpi pożyczka (CY=1)
        JP  NC, DZP  ;nie ma pożyczki, odejmuj dalej
        DEC BC       ;skoryguj wartość ilorazu
```

Zauważmy, że ostatnie, nieudane odejmowanie (to, po którym C=1) także jest zarejestrowane. „Unieważnimy" je, zmniejszając o 1 zawartość pary BC.

Pytanie 13. Jak uniknąć instrukcji DEC BC, tak, aby od razu po zakończeniu pętli wynik był poprawny?

Jeśli potrzebne jest szybkie mnożenie lub dzielenie, stosuje się efektywniejsze algorytmy. Mnożenie najczęściej realizuje się przez sumowanie przesuniętych względem siebie iloczynów częściowych — podobnie jak w odręcznych rachunkach na liczbach dziesiętnych. Poniższy program mnoży dwie liczby bez znaku: jednobajtową w A i dwubajtową w DE. Wynik jest trzybajtowy: najstarszy bajt w A, dwa młodsze w HL:

```
MNOZ21: LD   B,  8       ;ładuj licznik bitów w A
        LD   HL, 0       ;zeruj  wstępnie iloczyn
MNOZ22: ADD  HL, HL      ;przesuń  iloczyn w lewo
        ADC  A,  A       ;wsuń do A bit CY po ADD
        JP   NC, NIEDOD  ;skocz, gdy wysun. bit=0
        ADD  HL, DE      ;dodaj iloczyn częściowy
        ADC  A,  0       ;uwzględnij ewent. carry
NIEDOD: DEC  B           ;odlicz   kolejny   bit
        JP   NZ, MNOZ22  ;powtórz,gdy nie ostatni
```

W programie zastosowano chytrą sztuczkę: rozkaz ADC A, A jednocześnie wyprowadza do CY kolejne bity mnożnika (ze starszych pozycji) i wprowadza z prawej strony do A najstarsze bity iloczynu. Powyższy program w większości przypadków działa znacznie szybciej niż warianty z cyklicznym dodawaniem nieprzesuniętego mnożnika.

5. NOWE MECHANIZMY, NOWE PROBLEMY

Poznaliśmy już „abecadło" programowania w języku asemblera. Możemy więc zaryzykować spotkanie z mechanizmami szczególnie charakterystycznymi dla języka maszynowego, nie mającymi bezpośrednich odpowiedników w większości języków wysokiego poziomu.

5.1. Stos

Rejestr SP, tzw. wskaźnik stosu, był dotychczas tematem „tabu". Pojęcie stosu nie jest nam jednak obce. Studiujemy literaturę i wyjmujemy z regału jeden tom po drugim. Gdy znajdziemy pożyteczne informacje, odkładamy książkę na stosik, utworzony z poprzednio wyselekcjonowanych pozycji. Kończąc szperanie w bibliotece, dysponujemy stosem woluminów. Ostatnio położony leży na szczycie stosu i jest łatwo dostępny. Aby dotrzeć do książek położonych wcześniej, czyli znajdujących się niżej, musimy najpierw zdjąć ze stosu i przenieść w inne miejsce wszystkie tomy położone powyżej.

Stos jest świetnym sposobem czasowego przechowywania pewnych obiektów, gdyż jest prosty w obsłudze. Jedyne możliwe czynności to układanie, wysyłanie na stos i zdejmowanie, pobieranie ze stosu. Czynności te zawsze odbywają się na szczycie stosu, nie naruszając głębiej położonych warstw.

Mikroprocesor także potrafi „układać na stosie" i „zdejmować ze stosu" — oczywiście nie książki, lecz bajty. W pojedynczej operacji procesor może ułożyć na stosie lub odczytać z niego zawartość pary rejestrów: BC, DE, HL i AF. Ponieważ akumulator jest „bez pary", łączy się go z rejestrem stanu.

Stos procesora przypomina regał z półkami numerowanymi od dołu do góry (im półka położona wyżej, tym jej numer większy). Na każdej półce mieści się jeden bajt. SP to tabliczka z numerem najniższej zajętej półki (stos ustawiony jest „do góry nogami" tzn. półki powyżej SP są już wszystkie zajęte). Gdy trzeba „położyć" parę bajtów, ładują się one na dwóch najwyższych wolnych półkach, zaś zawartość SP jest zmniejszana o 2. Odwrotnie przy pobieraniu: zabierany jest bajt z półki wskazywanej przez SP i bajt z półki powyżej (SP+1), zaś SP jest zwiększany o 2. Pobranie bajtów ze stosu nie oznacza ich fizycznej likwidacji, lecz raczej skopiowanie, odczyt. Skopiowane bajty pozostają na swych „półkach", lecz półki te pozostają po odczycie poniżej wskaźnika SP. Oznacza to, że półki te są dostępne do ponownego wykorzystania: przy najbliższej okazji zostaną na nich umieszczone nowe bajty, rugując poprzednią zawartość.

Reasumując, półki o numerach niższych od SP są wolne, półki o numerach nie mniejszych od SP — zajęte. Zawartość zajętych półek jest chroniona przed zniszczeniem aż do chwili odczytu. SP wskazuje półkę, stanowiącą aktualny wierzchołek stosu. Położoną na

stosie wartość można odczytać tylko raz, gdyż po odczycie wskaźnik SP przesuwa się, wskazując na kolejną nieodczytaną półkę, położoną „piętro wyżej".

Oto rozkazy obsługujące stos, czyli układające i zdejmujące z niego pary bajtów:

```
PUSH  BC   ;ułóż na stosie zawartość pary BC
PUSH  DE   ;ułóż na stosie zawartość pary DE
PUSH  HL   ;ułóż na stosie zawartość pary HL
PUSH  AF   ;ułóż na stosie akumulator i bity stanu

POP   BC   ;skopiuj wierzchołek stosu do pary BC
POP   DE   ;skopiuj wierzchołek stosu do pary DE
POP   HL   ;skopiuj wierzchołek stosu do pary HL
POP   AF   ;skopiuj wierzchołek do A i rejestru stanu
```

Z wyjątkiem ostatniego, żaden z wymienionych rozkazów nie zmienia bitów stanu. Rozkaz POP AF odtwarza bity stanu, wysłane na stos poprzednim rozkazem PUSH AF.

Gdzie w PAO mieści się stos? Zależy to tylko od zawartości SP. Trzeba zarezerwować w pamięci obszar odpowiedniej wielkości, a następnie jego górny adres, powiększony o 1, wpisać do rejestru SP. Wykonuje to rozkaz LD SP, STAŁA (SP jest rejestrem szesnastobitowym):

```
LD  SP, 60000  ;wpisz do rejestru SP liczbę 60000
```

Po tym rozkazie stos będzie mieścił się w komórkach od 59999 w dół. Dlaczego nie od 60000? Powiedzieliśmy, że SP wskazuje na ostatnią zajętą komórkę. Zawartość SP jest więc zmniejszana o 2 przed układaniem nowej pary bajtów i zwiększana o 2 po zdejmowaniu. Na początku stos jest pusty, ani jedna komórka nie jest zajęta i SP wskazuje poza obszar stosu.

Najprostszym zastosowaniem stosu jest chwilowe przechowywanie zawartości rejestrów, co pozwala używać tych rejestrów do celów doraźnych. Należy oto zwiększyć zawartość słowa PAO pod adresem 30000 o 2048. Przydałyby się nam rejestry HL i BC, lecz zawierają one potrzebne dane i ich zawartości nie wolno naruszyć. Jak postąpić? Przechować HL i BC na stosie, a po skończonym dodawaniu — odtworzyć je:

```
PUSH HL              ;przechowaj HL na stosie
PUSH BC              ;przechowaj BC na stosie
LD   HL, (30000)     ;ładuj do HL słowo spod 30000
LD   BC, 2048        ;ładuj do BC stałą 2048
ADD  HL, BC          ;dodaj BC do zawartości HL
LD   (30000), HL     ;wpisz wynik pod adres 30000
POP  BC              ;odtwórz pierwotną zawartość BC
POP  HL              ;odtworz pierwotną zawartość HL
```

Porządek odczytu par rejestrów ze stanu jest odwrotny, niż kolejność ich układania. Następny przykład: trzeba dodać rejestr B do akumulatora, a przy zerowym wyniku skoczyć do etykiety ZERO. Przedtem jednak zawartość B powinna zostać zwiększona o 2:

```
ADD    B      ;dodaj rejestr B do akumulatora
PUSH   AF     ;zapamiętaj na stosie A i bity stanu
INC    B      ;zwiększ o 1 zawartość rejestru B
INC    B      ;zwieksz o 1 zawartość rejestru B
POP    AF     ;odtwórz akumulator i bity stanu
JP Z,  ZERO   ;skocz, jeśli bit Z=1
```

Instrukcje INC B nie wpływają na zawartość akumulatora, lecz zmieniają bity stanu. PUSH AF zapamiętuje na stosie bity stanu bezpośrednio po operacji dodawania. Rozkaz POP AF odczytuje ze stosu zapamiętaną wartość rejestru stanu i przywraca bitom warunków wartości sprzed pierwszego rozkazu INC B. Skok warunkowy wykona się więc tak, jak gdyby znajdował się bezpośrednio po rozkazie ADD B.

[illegible] Jak [illegible] zawartość BC i DE?

W bardziej zaawansowanych zastosowaniach mogą okazać się przydatne następujące operacje z udziałem wskaźnika stosu SP:

```
INC  SP     ;zwiększ  SP o 1 bez zmiany bitów stanu
DEC  SP     ;zmniejsz SP o 1 bez zmiany bitów stanu
LD   SP,HL  ;skopiuj  do SP  zawartość  pary  HL
ADD  HL,SP  ;do HL dodaj SP, zmieniając odpowiednio
            ;bit CY i zachowując resztę bitów stanu
```

[illegible] 15. W [illegible] można zapamiętać aktualną zawartość rejestru SP w PAO [illegible] np. 500?

5.2. Podprogramy

Rozkazy maszynowe są prymitywne. Nawet proste operacje, jak mnożenie czy dzielenie, potrzebują wielu rozkazów. Brakujące rozkazy można zastąpić wywołaniami podprogramów, podobnie jak np. w języku BASIC (instrukcje GOSUB i RETURN).

Podprogram maszynowy jest grupą rozkazów, stanowiących całość z funkcjonalnego punktu widzenia (wykonujących pewną wyodrębnioną czynność) i umieszczonych poza głównym nurtem programu. Podprogramy maszynowe są często nazywane procedurami maszynowymi. Wykonanie podprogramu odbywa się na żądanie, wyrażone rozkazem CALL. Po zrealizowaniu zadania kontynuowany jest program główny, czyli wykonuje się rozkaz następny za rozkazem CALL, który podprogram wywołał.

Rozkaz CALL jest odmianą skoku JP. Jedyna różnica polega na tym, że CALL przed wykonaniem skoku układa na stosie aktualną zawartość licznika programu PC. Innymi słowy, po CALL na szczycie stosu znajduje się adres rozkazu następnego za rozkazem CALL. Adres ten umożliwia powrót do miejsca wywołania, dlatego nazywany jest adresem powrotu:

```
CALL 45000  ;skocz do 45000, zapisując adres powrotu
```

Ostatnim rozkazem w podprogramie musi być rozkaz powrotu z podprogramu RET. Oznacza on także skok. Adres skoku nie jest jednak podany w samym rozkazie, lecz zostanie odczytany ze szczytu stosu. Ponieważ podprogram został wywołany rozkazem CALL, który pozostawił na szczycie stosu adres powrotu, to RET spowoduje skok do następnego rozkazu po CALL. CALL i RET traktują stos podobnie jak PUSH i POP.

Niech zadaniem programu będzie obliczenie wyrażenia: 10* (10*b+a) i niech wartość a znajduje się w parze rejestrów BC, zaś wartość b — w parze HL. Potrzebna będzie dwu-

krotnie operacja mnożenia przez 10. Sformułujemy w tym celu odpowiedni podprogram, mnożący przez 10 zawartość pary HL:

```
;MNOŻENIE LICZBY DWUBAJTOWEJ PRZEZ  10
;dane:    mnożona liczba w HL
;wyniki: iloczyn w HL  ;tracone:  HL i  bity stanu
MNOZ10: PUSH BC      ;przechowaj BC na stosie, gdyż
        LD B, H      ;BC będzie rejestrem roboczym
        LD C, L      ;skopiuj  zawartość  HL do BC
        ADD HL, HL   ;podwój zawartość  HL; HL= x*2
        ADD HL, HL   ;podwój zawartość  HL; HL= x*4
        ADD HL, BC   ;dodaj BC do HL; HL= x*4+x=x*5
        ADD HL, HL   ;podwój zawartość HL; HL= x*10
        POP BC       ;odtwórz  pierwotną  treść  BC
        RET          ;wróć do programu wywołującego
```

Podprogram jest mało uniwersalny, ale szybki, oprócz tego operuje zarówno na liczbach ze znakiem, jak i bez. Mnożenie przez 10 odbywa się według przepisu: 2* (4*x+x), gdzie x jest pierwotną zawartością HL. Rejestr BC spełnia funkcję pomocniczą: przechowuje pierwotną wartość liczby w HL. Program, wywołujący podprogram MNOZ10, „ma prawo" nie interesować się szczegółami działania podprogramu. Pierwotną treść rejestru roboczego BC należy więc przechować na stosie [PUSH BC] i odtworzyć przed powrotem do programu wywołującego [POP BC]. Postępowanie takie nie zawsze wydaje się niezbędne — np. wtedy, gdy program główny nie przechowuje w BC żadnej ważnej informacji. Często zdarza się jednak, że podprogram jest z biegiem czasu używany także w innych miejscach programu, niż pierwotnie planowano. Może się wtedy zdarzyć, że podprogram niechcący niszczy ważne dane. Polowanie na takie błędy bywa żmudne i frustrujące. Nic dziwnego: program w języku asemblera to tysiące szczegółów, zaś pamięć ludzka jest zawodna. Najlepiej więc konstruować podprogramy tak, aby nie zmieniały niczego, co nie wiąże się w bezpośredni sposób z zadaniem podprogramu, jak np. rejestry zawierające parametry lub rezultaty. Podprogram powinien być możliwie „przezroczysty", czyli program wywołujący nie powinien „zauważyć" żadnych ubocznych skutków spowodowanych przez podprogram, a nie związanych bezpośrednio z przekazywaniem parametrów do lub z podprogramu. Takie postępowanie uchroni nas przed przykrymi niespodziankami. Odstępstwo od zasady „przezroczystości" może być uzasadnione jedynie koniecznością optymalizacji czasu wykonania krytycznych fragmentów programu.

Każdy podprogram powinien mieć nagłówek, komentujący jego działanie, sposób przekazywania parametrów i wyników oraz ewentualne efekty uboczne. Raz opracowane podprogramy zechcemy zapewne wykorzystać w następnych programach i utworzymy „bibliotekę" sprawdzonych procedur, realizujących typowe operacje. Dobrze opisany podprogram można łatwo i bezbłędnie zastosować nawet wtedy, kiedy szczegóły związane z jego funkcjonowaniem dawno wywietrzeją nam z głowy. Dodatkowy wysiłek włożony w zapewnienie „przezroczystości" oraz sumienny opis to bardzo opłacalne inwestycje.

Zastosujmy wreszcie podprogram MNOZ10 w praktyce. Oto fragment programu głównego, wykonującego przepisane rachunki:

```
OBLICZ: CALL MNOZ10  ;mnóż HL*10          [10*b]
        ADD  HL, BC  ;do HL dodaj BC      [10*b+a]
        CALL MNOZ10  ;mnoż HL przez 10   [10*(10*b+a)]
```

Nietrudno zauważyć, że program główny zyskał na czytelności, gdyż nie zawiera zbędnych szczegółów, lecz tylko syntetyczne czynności, odpowiadające występującym we wzorze operacjom. Stosowanie podprogramów jest więc także doskonałym sposobem zwięk-

szenia przejrzystości programów i poprawy ich struktury (każdy podprogram realizuje wyodrębnioną czynność).

W pamięci ROM mikrokomputera znajduje się wiele użytecznych podprogramów, które można wykorzystać do własnych celów. Jednym z najważniejszych jest podprogram wyprowadzający na ekran pojedynczy znak. W ZX Spectrum wystarczy CALL 16. Kod znaku musi znajdować się w akumulatorze. Poniższy program wyświetli litery ABC:

```
LITERY: LD  A, 65   ;ładuj do akumulatora kod "A"
        CALL 16     ;wyprowadź  symbol  na  ekran
        LD  A, 66   ;ładuj do akumulatora kod "B"
        CALL 16     ;wyprowadź  symbol  na  ekran
        LD  A, 67   ;ładuj do akumulatora kod "C"
        CALL 16     ;wyprowadź  symbol  na  ekran
```

Liczby: 65, 66 i 67 to właśnie kody liter „A", „B" i „C". Dla niewtajemniczonych powyższy tekst może być niezłą łamigłówką. Zapiszmy go zatem inaczej, korzystając z oferowanych przez asembler udogodnień:

```
        PIZNAK EQU 16   ;zdefiniuj adres  PIZNAK
LITERY: LD  A,  'A'     ;ładuj do rej. A kod "A"
        CALL    PIZNAK  ;wyprowadź znak na ekran
        LD  A,  'B'     ;ładuj do rej. A kod "B"
        CALL    PIZNAK  ;wyprowadź znak na ekran
        LD  A,  'C'     ;ładuj do rej. A kod "C"
        CALL    PIZNAK  ;wyprowadź znak na ekran
```

Na początku zdefiniowaliśmy symbol PIZNAK, któremu została przypisana wartość 16. Od tej chwili przy każdym wystąpieniu PIZNAK asembler wstawi w to miejsce liczbę 16.

W innych komputerach operacja wyprowadzania znaku odbywa się inaczej. W języku BASIC wyświetlanie informacji odbywa się instrukcją PRINT, niezależnie od typu komputera. Programując w asemblerze musimy odwołać się do specyficznych właściwości komputera nawet przy elementarnych operacjach wejścia/wyjścia. Najlepszym rozwiązaniem będzie skupienie wszystkich szczegółów, związanych z konkretnym sprzętem, w jednym podprogramie. Jeśli program korzystający z operacji wejścia/wyjścia ma być adaptowany dla konkretnego komputera, wystarczy wymienić tylko procedurę sterującą w znormalizowany sposób konkretnym urządzeniem. Procedura obsługi urządzenia jest w informatycznym slangu określana najczęściej jako „handler" lub „driver". Niezależnie od sposobu działania procedury obsługi sposób przekazywania parametrów musi być oczywiście w każdym przypadku taki sam.

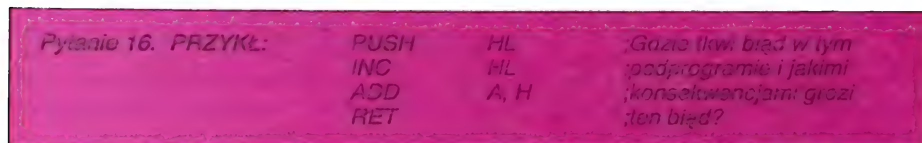
Możliwości techniczne rozmaitych typów mikrokomputerów mogą się istotnie różnić. W każdym przypadku dostępna jest jednak jedna podstawowa czynność: wyprowadzenie pojedynczego znaku na ekran. W przyszłych rozważaniach korzystać będziemy tylko z tej jednej operacji odwołującej się do możliwości sprzętowych. Odpowiedni podprogram (handler) nazwiemy (oznaczymy etykietą) PISZZN. Oto przykład definicji procedury PISZZN dla ZX Spectrum i komputerów z systemem CP/M:

```
;Dla ZX Spectrum        | ;Dla systemu CP/M 80
PISZZN: PUSH AF         | PISZZN: PUSH AF
        PUSH HL         |         PUSH HL
        PUSH DE         |         PUSH DE
        PUSH BC         |         PUSH BC
        CALL 16         |         LD   E, A
        POP  BC         |         LD   C, 2
```

```
POP   DE
POP   HL
POP   AF
RET
```

```
CALL 5
POP   BC
POP   DE
POP   HL
POP   AF
RET
```

PISZZN jest całkowicie przezroczysta (zachowuje nawet bity stanu sprzed wywołania). Kod wyprowadzanego znaku musi znajdować się w chwili wywołania w akumulatorze.



```
Pytanie 16.  PRZYKŁ:     PUSH    HL      ;Gdzie tkwi błąd w tym
                         INC     HL      ;podprogramie i jakimi
                         ADD     A, H    ;konsekwencjami grozi
                         RET             ;ten błąd?
```

5.3. Technika korzystania z podprogramów

Podprogramy są jednym z najważniejszych narzędzi programisty, a właściwa technika ich stosowania decyduje o stylu programowania. Głównym problemem jest przekazywanie parametrów. Najprościej czynić to za pośrednictwem rejestrów procesora lub stosu.

Często występującą czynnością jest wyprowadzanie napisów. Rozwiązania polegające na kolejnym ładowaniu do akumulatora kodów znaków rozkazem LD A, STAŁA i wywoływaniu procedury PISZZN trudno uznać za racjonalne. Przyjmijmy, że tekst przedstawiony jest w pamięci maszyny jako ciąg bajtów, przedstawiających kody kolejnych znaków. Koniec tekstu jest sygnalizowany np. bajtem o wartości 0. Ten bajt o zarezerwowanej wartości spełnia rolę „czerwonej latarni" tekstu i jest często nazywany „wartownikiem" lub „strażnikiem". Zdefiniujemy podprogram wyświetlający tekst o dowolnej długości. Jedyny parametr: adres pierwszego bajtu tekstu, jest przekazywany w parze rejestrów HL:

```
;WYPROWADZANIE  TEKSTU  ZAKONCZONEGO  WARTOWNIKIEM
;dane:  adres tekstu w HL        (wartownik =bajt 0)
;       w chwili powrotu HL wskazuje adres wartownika
;       reszta  rejestrów  i  bity  stanu - bez zmian
TEKST1: PUSH  AF            ;przechowaj akumulator
PEKST1: LD  A, (HL)         ;pobierz kod kolejnego znaku
        AND A               ;testuj kod,ustaw bity stanu
        JP  Z, KEKST1       ;jeżeli zero,  zakończ pracę
        CALL  PISZZN        ;wyprowadź  znak  na  ekran
        INC  HL             ;ustaw adres następnego kodu
        JP  PEKST1          ;i powtórz pętlę

KEKST1: POP   AF            ;odtwórz  treść  akumulatora
        RET                 ;wróć  do miejsca  wywołania
```

Oto przykład zastosowania podprogramu TEKST1 do wyświetlenia krótkiego napisu:

```
NAPIS1: DEFB 'OTO NAPIS'  ;tu asembler wstawi  ciągi
NAPIS2: DEFB ' UWAGA! '   ;kodów     odpowiadających
        DEFB 0            ;podanym napisom  i bajt 0
```

```
WYSWI1: LD   HL, NAPIS1    ;ładuj do HL  adres NAPIS1
        CALL TEKST1        ;wyświetl tekst na ekranie
        LD   HL, NAPIS2    ;ładuj do HL  adres NAPIS2
        CALL TEKST1        ;wyświetl tekst na ekranie
```

Dwie ostatnie dyrektywy DEFB mogliśmy zastąpić jedną, z tym samym skutkiem.

```
NAPIS2: DB ' UWAGA! '  0
```

Wywołany podprogram TEKST1 odczyta bajt z komórki PAO wskazanej przez HL i wpisze go do akumulatora. Rozkaz AND A ustawi bity stanu bez zmiany zawartości akumulatora. Jeśli bit Z=1, oznacza to odczyty zera, czyli wartownika (wskaźnika końca tekstu). W tym przypadku nastąpi powrót. W przeciwnym razie znak zostanie wysłany na ekran, a zawartość HL powiększona o 1. HL wskazuje więc na kolejny bajt tekstu. Przy powtórzeniu pętli bajt ten zostanie zbadany itd. Ostatecznym efektem będzie wyświetlenie napisu: „OTO NAPIS UWAGA! UWAGA!". Zauważmy, że formalny zapis tekstu w dwóch deklaracjach DEFB nie ma żadnego wpływu na realizację programu. Gdy program znajdzie się w pamięci, jest już tylko jednolitą masą bajtów. Postać źródłowa jest bez znaczenia.

Główną wadą przedstawionego rozwiązania jest fakt, że teksty są umieszczone w innym miejscu, niż program, który żąda ich wyświetlenia. Nie ułatwia to lektury programu, czy nie prościej byłoby umieścić tekst razem z wywołaniem? Funkcjonuje to doskonale w językach wyższego poziomu, np. PRINT „OTO NAPIS". Ano, spróbujmy. Postać tekstu pozostanie taka, jak poprzednio, ale będziemy podawać go bezpośrednio za wywołaniem. Wyłaniają się dwa problemy: skąd podprogram będzie „wiedział", jaki jest adres tekstu, i jak rozwiązać sprawę powrotu? Powrót musi nastąpić bowiem nie do pierwszego bajtu za rozkazem CALL, lecz do pierwszego bajtu za wartownikiem wyświetlonego tekstu!

Skoro tekst zaczyna się bezpośrednio za rozkazem CALL, to jego adres jest po prostu umieszczonym na szczycie stosu adresem powrotu. Adres ten można jednak „zdjąć ze stosu", odczytując go np. do pary HL rozkazem POP HL:

```
;WYPROWADZANIE TEKSTU WPISANEGO W KOD PROGRAMU
;dane: tekst za wywołaniem, zamknięty bajtem 0
;HL niszczone,inne rej. i bity stanu bez zmian
TEKST2: POP HL          ;adres  powrotu  do  HL
        PUSH AF         ;przechowaj  akumulator
PEKST2: LD   A, (HL)    ;odczytaj kolejny  bajt
        INC  HL         ;ustaw adres następnego
        AND  A          ;testuj odczytany  bajt
        JP   Z,KEKST2   ;jeśli 0, zakończ pracę
        CALL PISZZN     ;wyświetl podany symbol
        JP   PEKST2     ;zbadaj  następny  bajt

KEKST2: POP  AF         ;odtwórz akumulator
        PUSH HL         ;zmodyfikowany    adres
        RET             ;powrotu na stos i wróć
```

Umieszczony w HL adres powrotu jest traktowany bezceremonialnie, jako zwykły wskaźnik kolejnego znaku. Para HL jest teraz inkrementowana zaraz po odczytaniu kodu znaku. To ważny szczegół: HL wskazuje zawsze adres następnego bajtu za aktualnie obrabianym. Po wykryciu wartownika para HL zawiera zatem adres pierwszego bajtu za wartownikiem, a więc właściwy adres powrotu. Jeśli zmodyfikowany adres wyślemy na stos, a potem spowodujemy wykonanie rozkazu RET, powrót nastąpi w pożądane miejsce

```
WYSWI2: CALL TEKST2        ;wywołaj procedurę TEKST2
        DEFB 'ABCDE', 0    ;tu asembler wstawi tekst
        LD  HL, 0          ;tutaj   nastapi   powrót
        ..........
```

Pytanie 17. Założyliśmy się z kolegą, że napiszemy program bez rozkazu RET. Podprogramy okazały się jednak niezbędne. Czy mamy szansę wygrać?

Procedura TEKST2 ma także wadę: niszczy zawartość pary HL. Łatwo jednak tego uniknąć używając rozkazu wymiany wierzchołka stosu z zawartością pary HL:

```
EX (SP), HL     ;zamień szczyt stosu z parą HL
```

Dwa bajty na szczycie stosu wędrują do pary HL, zaś ich miejsce zajmuje dotychczasowa zawartość pary HL. Bity stanu, wskaźnik stosu ani pozostałe rejestry nie ulegają zmianom. Zapiszemy ulepszony wariant podprogramu:

```
;WYPROWADZANIE TEKSTU WPISANEGO  W KOD PROGRAMU
;dane: brak; tekst musi być zamknięty  bajtem 0
;nie niszczone  żadne  rejestry  ani bity stanu
TEKST3: EX (SP), HL      ;adres do HL,HL na stos
        PUSH AF          ;przechowaj  akumulator
PEKST3: LD   A, (HL)     ;odczytaj kolejny  bajt
        INC  HL          ;ustaw adres następnego
        AND  A           ;testuj odczytany  bajt
        CALL NZ,PISZZN   ;jeśli nie 0,  wyświetl
        JP   NZ,PEKST3   ;jeśli nie 0,   powtórz
        POP  AF          ;odtwórz akumulator
        EX (SP), HL      ;odtwórz HL, ułóż adres
        RET              ;powrotu na stos i wróć
```

Tym razem procedura jest całkiem klarowna. Nietrudno zauważyć jeszcze jedną modyfikację: CALL NZ. Warunkowe wywołania podprogramów odpowiadają ściśle skokom warunkowym, podobnie jak bezwarunkowe wywołanie CALL bezwarunkowemu skokowi JP:

```
CALL NZ, ADRES1   ;wywołaj, jeśli Z =0 (nie zero)
CALL Z,  ADRES2   ;wywołaj, jeśli Z =1 (zero)
CALL NC, ADRES3   ;wywołaj, jeśli CY=0 (nie carry)
CALL C,  ADRES4   ;wywołaj, jeśli CY=1 (carry)
CALL P,  ADRES5   ;wywołaj, jeśli S =0 (plus)
CALL M,  ADRES6   ;wywołaj, jeśli S =1 (minus)
```

Wywołanie podprogramu nastąpi tylko przy spełnieniu podanego warunku. Wywołania warunkowe są cennym narzędziem programistycznym, pozwalając uniknąć części skoków JP, a tym samym poprawić strukturę programu (w naszym przykładzie także wyeliminowaliśmy jeden skok). To samo można powiedzieć o rozkazach warunkowego powrotu z podprogramu:

```
RET  NZ,    ;powróć, jeśli Z =0 (nie zero)
RET  Z,     ;powróć, jeśli Z =1 (zero)
RET  NC,    ;powróć, jeśli CY=0 (nie carry)
RET  C,     ;powróć, jeśli CY=1 (carry)
RET  P,     ;powróć, jeśli S =0 (plus)
RET  M,     ;powróć, jeśli S =1 (minus)
```

Powrót z podprogramu nastąpi tylko wtedy, gdy wybrany bit stanu znajduje się w żądanym stanie. W przeciwnym razie praca podprogramu będzie kontynuowana.

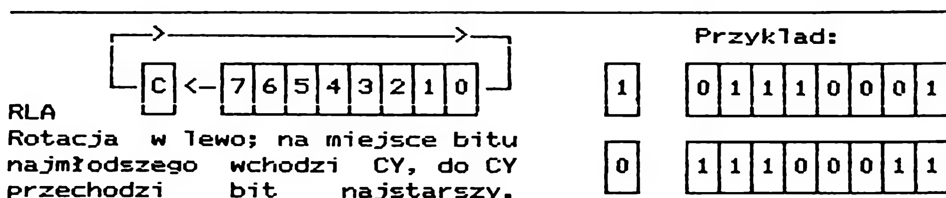
```
Pytanie 18: SIGN     AND  A           ;procedura SIGN wyznacza
                     JP   Z,KONIEC    ;dla akumulatora funkcję
                     JP   M,MINUS     ;znaku (0 dla A=0, +1 dla
                     LD   A,  1       ;A > 0, -1 dla A < 0). Wynik
                     RET              ;w akumulatorze. Jak
             MINUS:  LD   A,  -1      ;zapisać ten podprogram
             KONIEC: RET              ;bez użycia rozkazów JP?
```

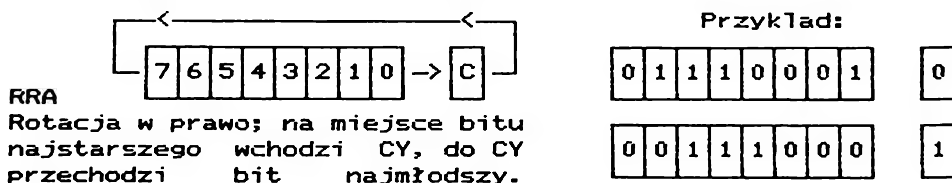
5.4. W lewo i w prawo

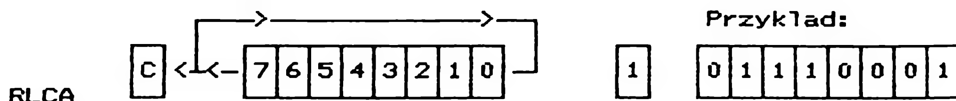
Podczas gdy w językach wyższego poziomu działamy zwykle na liczbach, operacje w języku maszynowym odbywają się często na bitach. Znajduje to odbicie w ubóstwie możliwości arytmetycznych i sporej ilości rozkazów logicznych. Zaliczają się do nich m.in. także rozkazy tzw. rotacji (przesunięcia cyklicznego) akumulatora: RRA, RLA, RRCA i RLCA. Powodują one przesunięcia wszystkich bitów akumulatora o jedną pozycję w lewo lub w prawo:

```
RLA     ;rotuj akumulator o 1 bit  w lewo
RRA     ;rotuj akumulator o 1 bit w prawo
RLCA    ;rotuj cyklicznie akumulator o 1 bit w  lewo
RRCA    ;rotuj cyklicznie akumulator o 1 bit w prawo
```

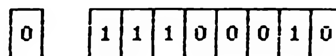
Działanie tych rozkazów ilustrują poniższe schematy i przykłady. W każdym przykładzie górny rysunek przedstawia stan akumulatora i bitu CY przed wykonaniem rozkazu, dolny — po wykonaniu:

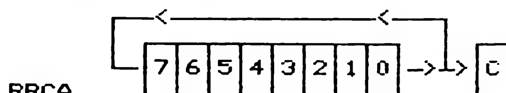




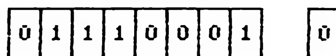


Rotacja cykliczna w lewo; bit najstarszy wchodzi równocześnie na miejsce najmłodszego i do CY.



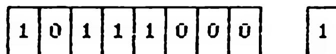


Przykład:



RRCA

Rotacja cykliczna w prawo; bit najmłodszy wchodzi równocześnie na miejsce najstarszego i do CY.



Rotacje cykliczne (RLCA i RRCA) różnią się od normalnych tylko tym, że bit CY nie pośredniczy w przesyłaniu wartości między najmłodszym i najstarszym bitem akumulatora, lecz tylko sygnalizuje tę wartość. Wszystkie cztery rozkazy rotacji wpływają wyłącznie na bit CY. Pozostałe bity stanu pozostają nienaruszone. Rotację umożliwiają różne operacje na pojedynczych bitach. Jak odwrócić kolejność bitów w akumulatorze — np. 10110111B — 11101101B? Potrzeba taka występuje np. w grafice komputerowej, przy konstruowaniu lustrzanych odbić. Realizuje to poniższy program.

```
ODWROT: LD   B, 8      ;rejestr B= licznik bitów
ODWP:   RRA            ;wysuń bit z lewej  do C
        LD   E, A      ;przechowaj wartość A w E
        LD   A, C      ;skopiuj rejestr C do  A
        RLA            ;wsuń bit C z prawej do A
        LD   C, A      ;wartość z powrotem  do C
        LD   A, E      ;odtwórz   zawartość   A
        DEC  B         ;zmniejsz  licznik  bitów
        JP   NZ, ODWP  ;jeśli nie zero,  powtórz
        LD   A, C      ;przenieś wynik do A
```

Pierwotna zawartość rejestru C jest bez znaczenia. W każdym powtórzeniu pętli z akumulatora „wysuwany" jest najmłodszy bit, który następnie „wsuwany" jest w miejsce najmłodszego bitu liczby przechowywanej w C. Ponieważ „wsuwanie" i „wysuwanie" odbywa się w przeciwnych kierunkach, to po ósmym cyklu kolejność bitów w C jest odwrotna, niż w pierwotnej zawartości akumulatora.

Pytanie 19. Czy w powyższym programie można bez zmiany jego funkcjonowania zamienić rozkaz RRA przez RRCA? Czy można zastąpić RLA przez RLCA?

Rozkazy rotacji pozwalają w prosty sposób analizować liczby dwójkowe bit po bicie. Wykorzystamy to, sporządzając podprogram wyświetlający na ekranie monitora układ bitów akumulatora w chwili jego wywołania:

```
;WYŚWIETLANIE  JEDNOBAJTOWEJ LICZBY DWÓJKOWEJ
;dane:     liczba do wyświetlenia w rejestrze  A
;niszczone tylko bity stanu,  rejestry bez zmian
BITYAK: PUSH BC       ;B i C  to rejestry robocze
        LD   B, 8     ;B  będzie licznikiem bitów
BIP:    RLCA          ;wsuń kolejny bit do  CARRY
```

```
        LD    C, A     ;przechowaj akumulator w  C
        LD    A, '0'   ;ładuj  do  A  kod cyfry  0
        JP    NC,ZER   ;Gdy CY= 0,   nie zmieniaj A
        LD    A, '1'   ;zmień kod cyfry '0' na '1'
ZER:    CALL  PISZZN   ;wyprowadź  właściwą  cyfrę
        LD    A, C     ;przesuwany bajt znowu do A
        DEC   B        ;zmniejsz o 1 licznik bitów
        JP    NZ,BIP   ;jeśli nie 0, powtórz pętlę
        POP   BC       ;odtwórz pierwotną treść BC
        RET            ;powrót do punktu wywołania
```

W każdym powtórzeniu pętli akumulator przesuwany jest o 1 bit w lewo. W ten sposób bity kolejno, od najstarszego do najmłodszego, kopiowane są do CY. Zależnie od tego, czy wartość bitu wynosi 0 lub 1, w chwili wywołania procedury PISZZN w akumulatorze znajdzie się kod cyfry „0" lub „1". Kod „0"=65, „1" =66; warto zauważyć, że instrukcję LD A, „1" można zastąpić przez INC A! Po ośmiokrotnym powtórzeniu rozkazu RLCA układ bitów w akumulatorze powraca do stanu początkowego. Dzięki temu w chwili powrotu zawartość akumulatora jest taka, jak w chwili wywołania.

Pytanie 20. Czy można zmodyfikować podprogram BITYAK tak, aby zawierał tylko jeden rozkaz skoku?

Gdybyśmy zechcieli wyprowadzić liczbę szesnastobitową, np. zawartość HL, wystarczyłoby dwukrotnie wywołać procedurę BITYAK:

```
BITYHL: LD    A, H     ;skopiuj starszy bajt do A
        CALL  BITYAK   ;wyprowadź  starszy   bajt
        LD    A, L     ;skopiuj młodszy bajt do A
        CALL  BITYAK   ;wyprowadź  młodszy   bajt
```

5.5. Rekursja

Większość programów prowadzi dialog z operatorem. Aby dialog ten był wygodny, komputer powinien wyświetlać liczby w postaci dziesiętnej. Jak przekształcić liczbę z postaci dwójkowej na dziesiętną? Możemy użyć metody kolejnego dzielenia przez 10. Reszta z pierwszego dzielenia będzie liczbą jednostek, reszta z drugiego — liczbą dziesiątek, z trzeciego — setek, itd. Zacznijmy od zdefiniowania uniwersalnej procedury, realizującej dzielenie liczb bez znaku (przyda się nam też do innych celów). Niech liczba w HL jest dzielona przez liczbę w DE. Po powrocie, w HL ma znaleźć się reszta z dzielenia, zaś w BC — iloraz. Użyjemy najprostszej techniki kolejnego odejmowania, nieco ją jednak ulepszając.

Aby uniknąć tysięcy odejmowań przy małym dzielniku, najpierw podzielimy przez dzielnik tylko starszy bajt dzielnej. Następnie uzyskaną resztę pomnożymy przez 256, dodamy młodszy bajt dzielnej i powtórzymy dzielenie. Z pierwszego dzielenia uzyskamy starszy, z drugiego — młodszy bajt ilorazu. W ten sposób liczba odejmowań nigdy nie przekroczy ok. 500, a w praktyce — kilkudziesięciu, co daje już dość przyzwoity czas wykonania:

```
;DZIELENIE Z RESZTĄ DWÓCH LICZB DWUBAJTOWYCH BEZ ZNAKU
;dane:    dzielna w HL, dzielnik w DE  (DE - zachowane)
;wyniki: reszta  w HL,  iloraz  w BC  (AF - niszczone)
```

```
DZIEL:   LD   B, L      ;przechowaj w B mł. bajt dzielnej
         LD   L, H      ;podziel HL przez 256, przenosząc
         LD   H, 0      ;starszy bajt do L i zerując H
         CALL DZIELB    ;podziel zawartość HL przez DE
         LD   H, L      ;mnóż HL razy 256,kopiując L do H
         LD   L, B      ;dodaj do HL mł. bajt dzielnej
         LD   B, C      ;kopiuj starszy bajt ilorazu do B
DZIELB:  LD   C, -1     ;C będzie licznikiem odejmowań
DZ1:     CALL ODHLDE    ;odejmij zawartość pary DE od HL
         INC  C         ;zwiększ o 1 licznik odejmowań
         JP   NC,DZ1    ;gdy wynik>=0,powtórz odejmowanie
         ADD  HL, DE    ;odtwórz HL sprzed ostatn. odejm.
         RET            ;powrót z wywołania
;ODEJMOWANIE DWÓCH LICZB DWUBAJTOWYCH
;dane: odjemna w HL, odjemnik w DE (BC i DE zachowane)
;wyniki: różnica w HL, bity stanu  (rej. A niszczony)
ODHLDE:  LD   A, L      ;odejmij w akumulatorze młodsze
         SUB  E         ;bajty, przenieś młodszy bajt
         LD   L  A      ;różnicy z powrotem do rejestru L
         LD   A, H      ;odejmij w akumulatorze starsze
         SBC  A, D      ;bajty, uwzględniając bit CY
         LD   H, A      ;kopiuj starszy bajt różnicy do H
         RET            ;powrót z wywołania
```

Przy okazji wzbogaciliśmy naszą bibliotekę podprogramów o procedurę odejmowania HL od DE. Korzysta z niej podprogram DZIEL, możemy jednak użyć jej też samodzielnie. Przenosząc zawartość H do L i zerując H dzielimy zawartość pary HL przez 256 (resztą jest pierwotna zawartość L). Podobnie, wpisując L do H i zerując L mnożymy HL przez 256. Ponieważ młodszy bajt HL jest zerem, dodanie do HL jednobajtowej liczby bez znaku sprowadza się do wpisania tej liczby do L. Zauważmy, że podprogram DZIELB jest po prostu fragmentem DZIEL, i że DZIEL wywołuje „własny ogon". Jest to przedsmak tego, co czeka nas za chwilę. Kolejne reszty z dzielenia przez 10 dają nam wartości cyfr, poczynając od najmłodszej. Tymczasem na ekran należy wysłać najpierw najstarszą cyfrę itd. Rozważmy prostszy przypadek: należy wyprowadzić tekst w odwrotnej kolejności: od ostatniego znaku do pierwszego. Oto odpowiedni podprogram wraz z przykładem wywołania:

```
;WYPROWADZANIE TEKSTU W ODWROTNEJ KOLEJNOŚCI
;dane: adres tekstu w HL; tekst kończy wartownik = 0
;po powrocie HL wskazuje następny bajt po wartowniku
;pozostałe rejestry i bajty stanu  zostają zachowane
WSPAK:   PUSH AF          ;ułóż na stos A i bity stanu
         LD   A, (HL)     ;pobierz kolejny znak tekstu
         INC  HL          ;ustaw adres następn. znaku
         AND  A           ;testuj odczytany kod znaku
         CALL NZ, WSPAK   ;jeśli nie wartownik,wywołaj
         CALL NZ,PISZZN   ;jeśli kod >0, wyświetl znak
         POP  AF          ;odtwórz A i rejestr stanu
         RET              ;wróć do miejsca wywołania

NAPISP:  DEFB 'PRZYKLAD TEKSTU', 0
START:   LD   HL, NAPISP
         CALL WSPAK
         ..........
```

Podprogram WSPAK wywołuje sam siebie. Taką technikę programowania nazywamy rekursją. WSPAK po wywołaniu pobiera pierwszy znak i bada, czy przypadkiem nie jest on wartownikiem. Jeśli tak, to następuje niezwłoczny powrót bez wyświetlenia znaku, poprzedzony odtworzeniem akumulatora i bitów stanu. Gdyby odczytany znak nie był wartownikiem, to przed wyświetleniem tego znaku podprogram zażąda wyświetlenia wszystkich następujących po nim, także w odwrotnym porządku. Jak? Po prostu wywoła zdefiniowany w tym właśnie celu podprogram WSPAK, czyli samego siebie! Działanie procedury WSPAK można opisać tak: „Pobierz znak. Jeśli nie jest on ostatnim znakiem tekstu, zastosuj WSPAK do pozostałej części tekstu, a potem wyświetl ten znak".

Procedury WSPAK „nie obchodzi", czy została wywołana z programu głównego, czy też ze swojego własnego wnętrza. Interesuje się ona tylko znakiem tekstu, wskazywanym przez HL. W ten sposób na stosie układane są ciągle nowe adresy powrotu i równocześnie wszystkie przeanalizowane do tej pory kody znaków. Ponieważ przy każdym wywołaniu adres w HL zwiększa się o 1, to za którymś razem zostanie napotkany wartownik i zamiast następnego wywołania nastąpi powrót. Jest to „kamień wywołujący lawinę": ze stosu pobierane są adresy i odbywają się powroty z kolejnych poziomów wywołań. Za każdym razem wyświetlany jest też znak o przechowanym na stosie kodzie. Znamy jednak odwracającą właściwość stosu: pierwszy ułożony znak będzie wyświetlony jako ostatni, zaś ostatni jako pierwszy. O to chodziło.

Amatorom języka BASIC trudno niekiedy oswoić się z istotą rekursji. Wbrew pozorom, rekursja odpowiada jednak naturalnej technice rozwiązywania wielu problemów. Ułatwia to pisanie dobrze zbudowanych programów. Mimo pewnych wad (duże zapotrzebowanie na pamięć stosu, mała efektywność z powodu czasochłonnych operacji na stosie) rekursja jest potężnym narzędziem programistycznym. Godnym uwagi jest fakt, że rekursja pozwala często ograniczyć liczbę skoków warunkowych i bezwarunkowych.



Wróćmy do programu konwersji dwójkowo-dziesiętnej. Tu także posłużymy się rekursją. Przygotujemy podprogram WYSDEC, sformułowany tak: „Podziel liczbę przez 10. Jeśli iloraz jest różny od 0, zastosuj do niego procedurę WYSDEC. Następnie niezależnie od ilorazu, wyświetl cyfrę odpowiadającą reszcie z dzielenia:"

```
;WYŚWIETLANIE W FORMIE DZIES. LICZBY DWUBAJT.BEZ ZNAKU
;dane:   wyprowadzana liczba w HL;
;        niszczone rejestry  A,B,C,D,E,H,L i bity stanu
WYSDEC:  LD   DE,10        ;załaduj dzielnik do pary DE
         CALL DZIEL        ;podziel zawartość HL przez 10
         LD   A, L         ;przenieś resztę do rejestru A
         LD   H, B         ;zaś iloraz przepisz do pary
         LD   L, C         ;HL na miejsce dzielnej
         PUSH AF           ;przechowaj resztę z dzielenia
         LD   A, L         ;oblicz iloczyn logiczny L i H
         OR   H            ;i ustal, czy zawartość HL = 0
         CALL NZ,WYSDEC    ;jeżeli nie, wywołaj WYSDEC
         POP  AF           ;ładuj do A resztę z dzielenia
```

```
        ADD  A, '0'        ;oblicz kod odpowiedniej cyfry
        CALL PISZZN        ;wyprowadź  tę cyfrę  na ekran
        RET                ;powrót z wywołania
```

Aby obliczyć kod cyfry, wystarczy do wartości cyfry (0 — 9) dodać kod cyfry „0". W ten sposób uzyskujemy kody 48 — 57, odpowiadające znakom „0" — „9". Zauważmy, że format wyświetlanej liczby przypomina trochę język BASIC: nieznaczące zera są usuwane, a reszta cyfr jest dosunięta do lewego marginesu.



5.6. Analiza danych wejściowych

Prawie każdy program domaga się od użytkownika jakichś danych, często liczbowych. Postarajmy się więc stworzyć podprogram, który ciąg cyfr dziesiętnych zamieni na liczbę dwójkową. Przyjmijmy, że ciąg cyfr dziesiętnych znajduje się już w PAO pod wskazanym adresem, dokąd został wprowadzony np. z klawiatury. Ogranicznikiem (sygnałem końca) liczby będzie pierwszy napotkany znak inny niż cyfra (spacja, „–", „+", CR, itd.):

Podprogram DECBIN wczytuje z bufora znak po znaku. Najpierw sprawdza się, czy znak jest cyfrą. Jeśli jego kod jest mniejszy od 48 lub większy od 57, to nie. Oznacza to napotkanie ogranicznika i powrót z wprowadzoną wartością w HL. Gdy znak jest cyfrą, jest ona „dopisywana" do liczby:

```
;KONWERSJA  LICZBY DZIESIĘTNEJ  NA DWOJKOWĄ DWUBAJTOWĄ
;dane:    adres początku danych  w  parze  rejestrów DE
;         ogranicznikiem jest każdy znak różny od cyfry
;wyniki:  wartość liczby w HL, ilość wprowadzonych cyfr
;         w C;po powrocie DE zawiera adres ogranicznika
;         rejestr A i bity stanu niszczone, B bez zmian
DECBIN:  LD   HL,0      ;zeruj HL -  wprowadzoną wartość
         LD   C, L      ;zeruj rejestr C -  licznik cyfr
DECBI1:  LD   A, (DE)   ;odczytaj z bufora kolejny  znak
         SUB  '9'+1     ;od A odejmij 58 -kod '9' plus 1
         RET  P         ;jeśli A>=0 powrót, bo nie cyfra
         ADD  10        ;dodaj do A 10 (-58+10 =48 ='0')
         RET  M         ;jeśli A<0, powrót, bo nie cyfra
         PUSH BC        ;przechowaj BC z licznikiem cyfr
         LD   C, L      ;skopiuj zawartość pary HL do BC
         LD   B, H
         ADD  HL, HL    ;pomnóż zawartość pary  HL przez
         ADD  HL, HL    ;10,  co  odpowiada przesunieciu
         ADD  HL, BC    ;dotychczas  wprowadzonych  cyfr
         ADD  HL, HL    ;dziesiętnych o 1 pozycję w lewo
         LD   C,  A     ;w  rejestrze  BC  umieść liczbę
         LD   B,  0     ;odpowiadającą odczytanej cyfrze
         ADD  HL, BC    ;i dodaj ją do pomnożonej liczby
         POP  BC        ;odtwórz licznik  cyfr w  rej. B
```

```
INC   C          ;odlicz wprowadzoną cyfrę
JP    DECBI1     ;zbadaj kolejny znak w buforze
```

Cyfry są pobierane od lewej. Kłopot w tym, że cyfr może być kilka. W chwili odczytu cyfry, nieznana jest jej waga, nie wiadomo, czy będzie to jedyna cyfra liczby jednocyfrowej, czy pierwsza cyfra czterocyfrowej. W pierwszym przypadku cyfra przedstawiałaby jednostki, w drugim — tysiące. Oto rozwiązanie: po rozpoznaniu kolejnej cyfry mnożymy całą dotychczas wprowadzoną wartość przez 10. Odpowiada to dziesięciokrotnemu powiększeniu wagi wszystkich wprowadzonych wcześniej cyfr. Np. w przypadku pierwszej cyfry liczby czterocyfrowej mnożenie przez 10 powtórzy się 3 razy — po rozpoznaniu cyfr: drugiej, trzeciej i czwartej. W efekcie odpowiada to mnożeniu przez 1000. Wprowadzona cyfra dodawana jest do pomnożenia liczby jako ilość jednostek i „awansuje" po wykryciu każdej następnej cyfry.

Jeśli zamiast pierwszej cyfry rozpoznany zostanie od razu ogranicznik, to nastąpi niezwłoczny powrót z wartością 0 w HL. Sytuację tę można rozpoznać, badając licznik cyfr w rejestrze C. W tym przypadku powinien wynosić 0. Nasza procedura jest prosta, lecz niedoskonała, nie wykrywa np. wprowadzenia liczby większej niż 65535. Nic nie stoi na przeszkodzie, aby ją ulepszyć...



Oprócz wczytywania liczb program musi często rozpoznawać zlecenia słowne. W najprostszym przypadku mogą one mieć postać pojedynczych liter. W zależności od podanego znaku powinien zostać zrealizowany jeden z kilku możliwych wariantów programu. Problem ten można oczywiście rozwiązać przy pomocy szeregu skoków warunkowych (zakładamy, że znak znajduje się w A):

```
CP   'A'          ;porównaj akumulator  ze wzorcem 'A'
JP   Z, ADRESA    ;jeśli zgodny,skocz do adresu ADRESA
CP   'S'          ;porównaj akumulator  ze wzorcem 'S'
JP   Z, ADRESS    ;jeśli zgodny,skocz do adresu ADRESS
...............
```

Gdy wariantów jest więcej, program przestaje być czytelny. Lepiej jest użyć ogólnej metody wyboru wariantów, a informacje o wzorcach znaków i odpowiadających im adresach skoków zgromadzić w oddzielnej strukturze danych, np. liście adresów, w wyodrębnionym miejscu programu. Niech program uwzględnia trzy możliwe warianty, oznaczone literami „A", „S" i „X". Przyjmijmy, że kod wariantu znajdzie się w rejestrze C. Gdyby wystąpił kod innego znaku niż dozwolone, należy wyświetlić komunikat. Lista wariantów składa się z trzybajtowych elementów: jednobajtowego wzorca (kodu znaku, odpowiadającego danemu wariantowi) oraz adresu skoku. Koniec listy jest sygnalizowany bajtem zerowym w miejsce kodu następnego wzorca:

```
SELEKT: LD    HL,ADRESY   ;załaduj adres listy wariantów do HL
SELEKP: LD    A, (HL)     ;pobierz  do  A kolejny bajt wzorca
        AND   A           ;ustaw bity stanu według  bajtu w A
        JP    Z, KONIEC   ;wzorzec = 0 - wskaźnik końca listy
        INC   HL          ;ustaw adres młodszego bajtu adresu
        CP    C           ;porównaj  wzorzec  ze  znakiem w C
        JP    NZ,NOWYZN   ;niezgodne - testuj  nast.  pozycję
        LD    A, (HL)     ;młodszy bajt  adresu  skoku  do  A
```

```
        INC   HL            ;ustaw w HL adres  starszego  bajtu
        LD    H, (HL)       ;starszy  bajt  adresu  skoku do HL
        LD    L, A          ;skompletuj w HL pełny  adres skoku
        JP    (HL)          ;skocz  do efektywnego adresu  w HL

NOWYZN: INC   HL            ;przeskocz   młodszy  bajt   adresu
        INC   HL            ;przeskocz   starszy  bajt   adresu
        JP    SELEKP        ;porównaj znak  z następnym wzorcem

KONIEC: CALL  TEKST2        ;wyprowadź komunikat o błędzie
        DEFB 'NIELEGALNY KOD', 0

        .............
ADRESY: DEFB   'A'          ;wzorzec  dla  pierwszego  wariantu
        DEFW   ADRESA       ;adres skoku dla pierwsz.  wariantu
        DEFB   'S'          ;dane dla drugiego  wariantu
        DEFW   ADRESS
        DEFB   'X'          ;dane dla trzeciego wariantu
        DEFW   ADRESX
        DEFB   0            ;wartownik - wskaźnik  końca  listy
```

Po pobraniu kodu wzorca program „przeskakuje" go, inkrementując HL. Gdy wzorzec jest zgodny ze znakiem w C, do HL jest ładowany adres z drugiego i trzeciego bajtu elementu listy (adres słowa, wskazywanego pierwotnie przez HL), po czym następuje skok do tego adresu. Gdyby wzorzec okazał się niezgodny z C, program przejdzie do analizy następnego elementu listy wariantów.

5.7. Komunikacja z otoczeniem

Komputer musi wymieniać informację z otoczeniem. W praktyce oznacza to, że komputerowi potrzebna jest możliwość odczytu poziomu logicznego na określonych końcówkach i ustawiania tego poziomu na innych wyprowadzeniach. Poziomowi logicznemu „0" odpowiada najczęściej napięcie zbliżone do 0 V, zaś „1" — napięcie zbliżone do 5 V, chociaż zdarzają się i inne konwencje. Do komunikacji z otoczeniem służą specjalne układy peryferyjne. Część ich wyprowadzeń jest połączona (bezpośrednio lub pośrednio) z procesorem, pozostałe są dołączane do różnych urządzeń zewnętrznych.

Procesor „porozumiewa się" z układami peryferyjnymi za pomocą tzw. rejestrów wejścia/wyjścia (rejestrów I/O), zwanych potocznie portami (bramami). Aby wyprowadzić dane na zewnątrz (np. na monitor), komputer wpisuje bajt do właściwego portu wyjściowego, w wyniku czego końcówki układu peryferyjnego przyjmują odpowiednie poziomy logiczne. Jeśli do końcówek tych są dołączone odpowiednie urządzenia, np. wzmacniacze, przekaźniki itd., procesor może sterować procesami zachodzącymi w jego otoczeniu. Żeby odczytać poziomy logiczne na grupie końcówek wejściowych układu peryferyjnego, procesor po prostu odczytuje właściwy port wejściowy.

W typowej konfiguracji, komputer z procesorami 8080 lub Z80 może mieć do 256 (numeracja 0 — 255) portów wejściowych i tyleż wyjściowych. Do ich obsługi służą specjalne rozkazy IN i OUT. Co prawda między komórkami PAO i portami istnieje pewne podobieństwo (odczyt i zapis bajtów), ale są i istotne różnice. Przede wszystkim inna jest fizyczna lokalizacja rejestrów i portów. Poza tym pewne rejestry służą wyłącznie do zapisu, zaś inne — tylko do odczytu. Może się zdarzyć, że np. rejestr wejściowy nr 1 i rejestr wyjściowy nr 1 będą

zlokalizowane w różnych układach peryferyjnych i służą do przekazywania zupełnie innych danych.

Rozkaz IN wprowadza do akumulatora bajt z portu wejściowego o podanym numerze. Rozkaz OUT wpisuje zawartość akumulatora do podanego portu wyjściowego. Obydwa rozkazy nie wpływają na bity stanu (podobnie jak rozkazy LD):

```
IN    A,   (64)     ;wprowadź do A bajt z portu nr 64
OUT   (3), A        ;zawartość  A wpisz do portu nr 3
```

Przypuśćmy, że zamierzamy wykorzystać komputer do sterowania girlandą choinkową. Lampki zostały podzielone na osiem grup, każda z nich włączana oddzielnym tyrystorem. Poszczególne tyrystory sterowane są z końcówek układu peryferyjnego, sterowanych przez port nr 8. Logiczny poziom „0" oznacza wyłączenie, „1" — włączenie grupy lampek. Program ma cyklicznie włączać i wyłączać poszczególne grupy lampek. Jednocześnie mają palić się cztery grupy. W każdym cyklu zapalana zostaje jedna z grup dotychczas wygaszonych, zaś jedna z grup zapalonych gaśnie. Wygaszanie i zapalanie każdej z grup ma odbywać się co 4 cykle. W ten sposób można uzyskać efekt „wędrującego światła":

```
ZWLOKA  EQU  20000           ;definicja stałej opóźnienia
        ORG  40000
START:  LD   A,  00001111B   ;wpisz do A kompozycję bitów
CYKL:   OUT  (8),A           ;wyprowadź  ją do portu nr 8
        RLCA                 ;rotuj rej. A o 1 bit w lewo
        LD   BC, ZWLOKA      ;ładuj  licznik   opóźnienia
OPOZN:  DEC  C               ;dekrementuj   młodszy  bajt
        JP   NZ, OPOZN       ;aż  do  chwili  wyzerowania
        DEC  B               ;dekrementuj starszy bajt aż
        JP   NZ, OPOZN       ;do  chwili  wyzerowania  BC
        JP   CYKL            ;powtórz cykl
```

Akumulator zawiera w każdej chwili cztery zera i tyleż jedynek. W każdym cyklu układ bitów przesuwany jest jednak o jedną pozycję w lewo, przy czym najstarszy bit wchodzi na miejsce najmłodszego. Pętla opóźniająca zapewnia niezbędną zwłokę rzędu 0,1 s (opóźnienie można regulować zmieniając stałą ZWLOKA). Zmieniając początkowy układ bitów w akumulatorze można programować inne ciekawe efekty.

Kolejny program realizuje prosty algorytm automatycznej kontroli i sterowania. Komputer nadzoruje układ ośmiu czujników temperatury, których stan jest odczytywany za pośrednictwem portu wejściowego nr 16. Każdemu czujnikowi jest przyporządkowany jeden bit; wartość 1 oznacza przekroczenie dozwolonej temperatury. Za pośrednictwem portu wyjściowego nr 17 komputer steruje sygnałem ostrzegawczym i wyłącznikiem awaryjnym. Ustawienie bitu nr 0 włącza sygnał, bitu nr 7 — powoduje awaryjne wyłączenie. Jeśli którykolwiek z czujników wskaże przekroczenie temperatury, powinien zostać włączony sygnał alarmowy. Gdy liczba uaktywnionych czujników wyniesie 3 lub więcej, musi nastąpić awaryjne wyłączenie:

```
        ORG  40100
NADZOR: IN   A,   (16)      ;wczytaj do A stan czujników
        AND  A              ;ustaw  bity stanu według  A
        JP   Z,   NADZOR    ;jeśli bity =0, testuj dalej
        LD   A,   1         ;co najmniej jeden z bitów=1
        OUT  (17), A        ;a więc  włącz sygnał alarmu
        LD   C,   0         ;zeruj C -  licznik  jedynek
NASTBI: ADD  A,   A         ;testuj kolejny bit i stan A
        JP   NC,  TENNIE    ;jeśli CY=0, bit  był zerowy
```

```
         INC   BC          ;gdy bit=1, zwiększ  licznik
TENNIE:  JP    NZ, NASTBI  ;gdy A>0 kontynuuj zliczanie
         LD    A, C        ;wpisz  licznik jedynek do A
         CP    2           ;porównaj  z wartością maks.
         JP    P, NADZOR   ;jeżeli  A<=2, testuj  dalej
         LD    A, 129      ;ładuj  do  A bajt 10000001B
         OUT   (17), A     ;uruchom sygnał  i  wyłącznik
         HALT              ;zatrzymaj  pracę  programu
```

Zliczanie bitów o wartości 1 odbywa się następująco. Rozkaz ADD A, A odpowiada podwojeniu zawartości A, inaczej — przesunięciu jego zawartości w lewo o 1 bit. Najstarszy bit przechodzi do CY, w miejsce najmłodszego wchodzi 0. Po ADD A, A uzyskujemy równocześnie dwie informacje: czy wysunięty właśnie bit był jedynką (CY=1) oraz czy pozostały jeszcze jakieś niezarejestrowane jedynki w A (Z=1). Najpóźniej po ośmiu powtórzeniach akumulator zostanie wyzerowany. Jeśli CY=1, trzeba inkrementować licznik. Dlaczego używamy INC BC zamiast INC C? Otóż INC BC nie zmienia bitów stanu, w szczególności bitu Z. Poza tym działanie obu rozkazów jest w tym przypadku identyczne (wartość C nigdy nie przekroczy 255, a więc rejestr B pozostanie niezmieniony). Dzięki temu po INC BC bit Z zachowuje wartość, jaką nadał mu rozkaz ADD A, A, możemy więc bity Z wykorzystać do sprawdzenia, czy jest sens zliczać dalej.

5.8. Obsługa przerwań

Przerwania są potężnym narzędziem, pozwalającym procesorowi szybko reagować na różne wydarzenia, a nawet pozornie równocześnie wykonywać kilka odrębnych zadań. Przerwanie zewnętrzne polega na tym, że do specjalnej końcówki mikroprocesora doprowadzany jest z zewnątrz sygnał przerwania. W Z80 polega on na zmianie poziomu logicznego z 1 na 0. W tym momencie procesor przerywa realizację bieżącego programu i wywołuje specjalnie przygotowany na tę okazję podprogram obsługi przerwania. Po wykonaniu odpowiednich czynności następuje powrót z podprogramu, po czym procesor, jak gdyby nic nie zaszło, kontynuuje przerwany program. Z80 ma dwie końcówki przerwań: INT i NMI, zaś 8080 — tylko INT.

NMI to wejście tzw. przerwań niemaskowalnych, tzn. takich, od obsługi których procesor nie może się „wykręcić". Po zmianie poziomu z 1 na 0 procesor wywołuje podprogram pod adresem 66H, czyli 102. Procedura obsługi przerwania niemaskowalnego musi być zamknięta specjalnym rozkazem RETN.

Przerwanie maskowalne INT działa jak NMI z dwoma wyjątkami: wywołany może zostać jeden z wielu podprogramów obsługi, zaś przerwaniu można zapobiec. Kiedy program z pewnych powodów nie może być przerwanym (np. przy odliczaniu czasu czy innych krytycznych czynnościach), może zablokować na pewien czas przerwania INT, wykonując rozkaz DI. Od tej chwili procesor nie reaguje na żadne wydarzenia na końcówce INT. Możliwość przerwań INT przywraca rozkaz EI. Skąd procesor „wie", pod jakim adresem znajduje się procedura obsługi przerwania? Nie wnikając w szczegóły, urządzenie zewnętrzne, które wysłało sygnał INT, dostarcza procesorowi dodatkowych informacji. Z80 rozróżnia tryby wyboru obsługi przerwania, wybierane specjalnymi rozkazami IM0, IM1 lub IM2. W trybie 0 (po IM0) Z80 reaguje na przerwania identycznie, jak 8080. Procesor potwierdza przyjęcie przerwania specjalnym sygnałem. Urządzenie, które spowodowało przerwanie, dostarcza w tym momencie jednobajtowego wskaźnika, wskazującego na jeden z ośmiu możliwych adresów: 0, 8, 16,..., 56. W najprostszym trybie 1 (po IM1) każdy sygnał przerwania INT powoduje wywołanie procedury obsługi pod adresem 38H czyli 56. 8080 pracujący w najprostszej konfiguracji może także reagować podobnie na INT. Tryb 2, tzw. wektoryzowany, umo-

żliwia wskazanie do 128 różnych procedur obsługi przerwania (w zależności od przyczyny).

W chwili przyjęcia przerwania INT, tzn. skoku do programu obsługi, automatycznie blokowana jest możliwość przerwań, jak po rozkazie DI. Zapobiega to przerwaniu programu obsługi przez inne przerwanie maskowalne. Dlatego bezpośrednio przed powrotem z obsługi przerwania należy wykonać rozkaz EI. Jego osobliwością jest fakt, że możliwość przerwań włączana jest z opóźnieniem o 1 rozkaz (po zakończeniu realizacji rozkazu następującego po EI). Jeśli po EI następuje RET, końcówka INT będzie odblokowana dopiero po powrocie z procedury obsługi przerwania. Przerwanie NMI także blokuje możliwość przerwań maskowalnych. Rozkaz RETN odtwarza stan sprzed wystąpienia NMI. Jeśli przerwania maskowalne były wówczas dozwolone, po RETN będą one ponownie odblokowane. Przed RETN nie jest więc potrzebny rozkaz EI (obsługę NMI można zakończyć także zwykłym rozkazem RET, lecz wówczas nie nastąpi automatyczne odtworzenie „stanu wrażliwości" procesora na przerwanie INT). Przerwanie NMI może w każdej chwili zawiesić obsługę przerwania maskowalnego INT, zaś w chwili obsługi NMI przerwania INT nie są przyjmowane. Mówimy, że NMI ma wyższy priorytet obsługi, niż INT.

Przerwanie polega na zawieszeniu, „uśpieniu" realizowanego programu na sygnał z zewnątrz i zatrudnieniu procesora na pewien czas do pilniejszych prac. Przerwanie przypomina nieco wywołanie podprogramu maszynowego, z tym, że podprogram wywoływany jest nie na życzenie programu, lecz na wyraźne żądanie z zewnątrz komputera, wyrażone zamianą poziomu logicznego na odpowiedniej końcówce. O ile podprogram „świadczy usługi" na rzecz programu, który go wywołał, otrzymuje od niego i przekazuje mu parametry, to program obsługi przerwania może nie mieć nic wspólnego z przerwanym programem. Po zakończeniu obsługi przerwania stan procesora musi być identyczny, jak w chwili przyjęcia przerwania. Po „przebudzeniu z narkozy" przerwany program nie może „zauważyć", że w międzyczasie coś się zmieniło, np. zawartość rejestrów lub bity stanu. Dlatego podstawowym obowiązkiem procedury obsługi przerwania jest przechowanie na stosie rejestru stanu (PUSH AF) i tych rejestrów, które będą wykorzystane przez procedurę. Rejestry te należy oczywiście odtworzyć bezpośrednio przed powrotem z procedury przerwania.

Przerwania zewnętrzne są bardzo często stosowane do odliczania czasu. Urządzenie zewnętrzne wysyła w regularnych odstępach czasu (np. co 20 ms, czyli 50 razy na sekundę) sygnał przerwania. Jedynym zadaniem programu obsługi jest zwiększyć po każdym przerwaniu o 1 zawartość licznika, utworzonego z kilku komórek PAO. Co sekundę licznik powiększany jest o 50, niezależnie od innych działań procesora. Odczytując licznik w różnych chwilach, program może zorientować się w upływie czasu. Napiszmy procedurę obsługi takiego przerwania, tzw. zegarowego. Niech przerwanie zegarowe jest jedyną możliwą przyczyną wystąpienia sygnału INT, a skok do procedury obsługi odbywa się pod adres 38H (56). Licznik składa się z trzech bajtów, umieszczonych poczynając od adresu 60000 (bajt najmłodszy):

```
LICZN    EQU  60000        ;adres licznika = 60000
         ORG  56
ZEGAR:   PUSH AF           ;przechowaj rejestr stanu
         PUSH HL           ;oraz zawartość pary HL
         LD   HL, LICZN    ;wpisz adres licznika do HL
         INC  (HL)         ;zwiększ o 1 najmłodszy bajt
         JP   NZ, ZEGAR1   ;nie ma przepełnienia-powrót
         INC  HL           ;ustaw adres starszego bajtu
         INC  (HL)         ;inkrementuj starszy bajt
         JP   NZ, ZEGAR1   ;nie ma przepełnienia-powrót
         INC  HL           ;HL=adres najstarszego bajtu
         INC  (HL)         ;inkrementuj najstarszy bajt
ZEGAR1:  POP  HL           ;odtwórz pierwotną treść HL
```

```
POP  AF          ;odtwórz rejestr stanu
EI               ;uaktywnij wejście INT
RET              ;powrót z obsługi przerwania
```

Procedura przechowuje rejestr stanu i parę HL, która będzie wykorzystana dla jej własnych potrzeb. Pozostałych rejestrów nie trzeba wysyłać na stos, gdyż ich zawartość nie zostanie zmieniona. Do HL ładowany jest adres najmłodszego bajtu po inkrementacji wartość bajtu jest różna od 0, może nastąpić powrót z obsługi. W przeciwnym razie oznacza to przepełnienie najmłodszego bajtu — trzeba zwiększyć o 1 bajt starszy. Jeśli i ten się przepełni, zostanie zwiększony bajt najstarszy. Nasz licznik jest 24 — bitowy, a zatem jego pojemność wynosi 2↑24−1=16777215 impulsów, czyli prawie 335544 sekund.

5.9. Przerwania programowe

Zarówno Z80, jak i 8080 dysponują ośmioma jednobajtowymi rozkazami wywołania podprogramów RST 0... RST 56, niekiedy nazywanymi rozkazami przerwań programowych albo restartu. Nie zawierają one adresu wywołania, gdyż każdy z nich uruchamia podprogram pod z góry zadanym adresem. Dla RST 0 jest to adres 0, dla RST 8 — 8, RST 56 — 56. Są to te same adresy, pod którymi umieszczane są procedury obsługi przerwań w procesorze 8080 lub Z80 w trybie 0. Wykonanie rozkazu RST powoduje więc identyczny skutek jak odpowiednie przerwanie INT, z wyjątkiem zablokowania linii przerwań.

Rozsądne użycie przerwań programowych może zwiększyć efektywność programu i wygodę programowania. Jeśli nie wykorzystujemy adresów 0...56 dla obsługi przerwań zewnętrznych, można tam zlokalizować najczęściej używane podprogramy. Podprogramy te wywołujemy następnie rozkazami RST, a nie CALL, co daje oszczędność zarówno pamięci, jak i czasu wykonania. Rozkaz RST używany jest często do wywołania różnych funkcji usługowych systemu operacyjnego. Co robić jednak, gdy możliwych funkcji jest więcej niż 8? Przed wywołaniem można załadować np. do akumulatora dodatkowy kod, który pozwoli zidentyfikować funkcję:

```
LD   A, 2        ;załaduj do A kod funkcji
RST  56          ;wywołaj program usługowy
```

Pierwszą czynnością podprogramu jest zbadanie kodu i skok do właściwego fragmentu, realizującego daną funkcję, np.:

```
         ORG  56
USLUGI:  ADD  A, A          ;oblicz przesunięcie w tablicy
         LD   C, A          ;młod. bajt przesunięcia do C
         LD   B, 0          ;starszy bajt przesunięcia = 0
         LD   HL, ADRESY    ;adres tablicy adresów do HL
         ADD  HL, BC        ;wyznacz adres adresu programu
         LD   A, (HL)       ;wpisz do A młod. bajt adresu
         INC  HL            ;ustaw w HL adres st. bajtu
         LD   H, (HL)       ;st. bajt adresu programu do H
         LD   L, A          ;młodszy bajt do L; HL = adres
         JP   (HL)          ;skocz do wyznaczonego adresu

ADRESY:  DEFW FUNKC0        ;adr. programu real. funkcję 0
         DEFW FUNKC1        ;adr. programu real. funkcję 1
         DEFW FUNKC2        ;adr. programu real. funkcję 2
```

```
DEFW FUNKC3        ;adr. programu real. funkcję 3
..........
```

Technikę wyboru wariantów poznaliśmy już wcześniej. Kod w akumulatorze wskazuje tę pozycję tablicy ADRESY, która zawiera właściwy adres programu realizującego wskazaną funkcję. Ponieważ pozycje tablicy mają po dwa bajty, kod w A trzeba podwoić (ADD A, A). Uzyskany adres względny (tzw. przesunięcie względem początku tablicy) wpisujemy do BC i dodajemy do adresu początku tablicy, uzyskując adres adresu programu. Następnie zamieniamy w HL adres adresu na adres efektywny i wykonujemy do niego skok.

Jeszcze lepszym sposobem jest umieszczenie jednobajtowego kodu funkcji bezpośrednio po rozkazie RST (metodę tą można oczywiście stosować także po CALL):

```
RST    56     ;wywołaj podpr. pod adresem 56
DEFB   2      ;kod funkcji    (funkcja nr 2)
```

Wywołany program obsługi musi odczytać ten bajt i „przeskoczyć" go, zwiększając o 1 adres powrotu. Dalszy tok postępowania — bez zmian:

```
         ORG  56
USLUGI:  EX   (SP), HL    ;adres kodu  funkcji do HL
         LD   A,   (HL)   ;odczytaj do A kod funkcji
         INC  HL          ;zwiększ adres powrotu o 1
         EX   (SP), HL    ;przenieś go znowu na stos
         ADD  A, A        ;wyznacz przesunięcie
         ............
```

6. UZUPEŁNIENIE WIADOMOŚCI O PROCESORACH 8080 i Z80

Dotychczas posługiwaliśmy się nieco uproszczonym, choć wystarczającym dla naszych potrzeb wyobrażeniem o procesorach 8080 i Z80. Czas uzupełnić zaległości, chociażby pobieżnie.

Oprócz bitów stanu S, Z i CY procesor 8080 ma jeszcze dwa, zaś Z80 — trzy inne bity stanu. Po operacjach logicznych bit P/V jest ustawiony, gdy wynik zawierał parzystą liczbę jedynek (np. 01110100B). Po operacjach arytmetycznych w procesorze 8080 jest podobnie, lecz Z80 interpretuje wtedy bit P/V jako wskaźnik nadmiaru arytmetycznego (P/V=1, gdy wystąpił nadmiar). Nadmiar występuje wtedy, gdy znak wyniku jest inny, niż wynikałby ze znaku operandów, np. ujemny po dodawaniu dwóch liczb dodatnich. Nadmiar arytmetyczny nie ma nic wspólnego z przeniesieniem. Różna interpretacja bitu P/V w przypadku rozkazów arytmetycznych jest jedynym odstępstwem od zgodności programowej procesora Z80 z 8080. Oto rozkazy warunkowe korzystające z bitu P/V:

```
JP   PO, ADRES1  ;skocz,   gdy P/V= 0
JP   PE, ADRES2  ;skocz,   gdy P/V= 1
CALL PO, ADRES3  ;wywołaj, gdy P/V= 0
CALL PE, ADRES4  ;wywołaj, gdy P/V= 1
RET  PO          ;powrót,  gdy P/V= 0
RET  PE          ;powrót,  gdy P/V= 1
```

Pozostałe dwa bity: H i N (w 8080 tylko H) są wykorzystywane wyłącznie przez rozkaz DAA, używany przy operacjach na liczbach w tzw. formacie BCD (cztery starsze i cztery młodsze bity — tzw. półbajty, przedstawiają po jednej cyfrze dziesiętnej). H jest tzw. przeniesieniem połówkowym (przeniesieniem z bitu nr 3), N wskazuje, czy ostatnią operacją arytmetyczną było dodawanie lub odejmowanie. Rozkaz DAA, użyty po dodawaniu lub odejmowaniu liczb BCD (w 8080 — tylko po dodawaniu), powoduje, że wynik także jest liczbą typu BCD.

Zarówno 8080, jak Z80 mogą bezpośrednio wpływać na bit CY, bez zmiany innych bitów stanu. Z80 ma rozkaz negacji arytmetycznej akumulatora:

```
SCF    ;nadaj bitowi CY wartość 1
CCF    ;neguj bit CY (zmień wartość na przeciwną)
NEG    ;neguj arytmet. A  bez zmiany  bitów stanu
```

Rozkaz NOP nie wykonuje żadnej operacji i nie zmienia bitów stanu. Mimo że pozornie bezużyteczny, przydaje się przy organizacji opóźnień czasowych i przy uruchamianiu programów.

Następne rozkazy odnoszą się wyłącznie do Z80. Procesor ten posiada dwa identyczne zestawy rejestrów roboczych. W każdej chwili aktywny jest tylko jeden, lecz mogą one być wymieniane pojedynczymi rozkazami:

```
EX   AF, AF'   ;zamień akumulator i rejestr stanu
EXX            ;zamień rejestry  B, C, D, E, H, L
```

„Ukryte" rejestry zachowują swą zawartość. Rozkazy EX AF, AF i EXX stanowią więc pewną alternatywę dla PUSH i POP, tym bardziej, że są wykonywane znacznie szybciej.

Z80 dysponuje sześcioma rozkazami skoków tzw. względnych. Są one dwubajtowe: adres skoku podany jest w kodzie rozkazu nie wprost, lecz jako różnica między adresem rozkazu skoku powiększonym o 2 (adresem następnego rozkazu), a właściwym adresem skoku. Rozkazy te pozwalają wykonywać skoki na odległość −128:+127 bajtów:

```
JR       ADRES   ;skocz  bezwarunkowo
JR   NZ, ADRES   ;skocz, gdy bit Z= 0
JR   Z,  ADRES   ;skocz, gdy bit Z= 1
JR   NC, ADRES   ;skocz, gdy bit CY=0
JR   C,  ADRES   ;skocz, gdy bit CY=1
DJNZ     ADRES   ;dekrementuj B i skocz, gdy wynik=0
```

Rozkaz DJNZ jest szczególnie użyteczny przy organizacji pętli. Odpowiada on parze rozkazów: DEC B i JR NZ, ADRES. Różnica polega na tym, że przy dekrementacji B nie są zmieniane bity stanu. Dzięki temu „administracyjna" czynność, jaką jest odliczanie powtórzeń, nie wpływa na przebieg pętli, co pozwala swobodniej dysponować bitami stanu.

Z80 pozwala bezpośrednio przesyłać dwubajtowe słowa między PAO, a dowolną parą rejestrów:

```
LD   (ADRES), BC   ;zapisz zawartość BC w PAO
LD   (ADRES), DE   ;zapisz zawartość DE w PAO
LD   (ADRES), SP   ;zapisz zawartość SP w PAO
LD   BC, (ADRES)   ;wpisz do BC zawartość słowa z PAO
LD   DE, (ADRES)   ;wpisz do BC zawartość słowa z PAO
LD   SP, (ADRES)   ;wpisz do SP zawartość słowa z PAO
```

Z80 posiada dwa szesnastobitowe, równoprawne rejestry indeksowe IX i IY. Służą one głównie do pośredniej adresacji komórek PAO. Efektywny adres komórki tworzy suma zawartości rejestru indeksowego i jednobajtowego przesunięcia, traktowanego jak liczba ze znakiem (–128:+127), co zapisujemy jako (IX+12) lub (IY−99). Adresacja z użyciem IX i IY jest dozwolona we wszystkich rozkazach, korzystających z adresacji pośredniej zawartością pary HL:

```
LD   A, (IX+1)    ;załaduj do akumulatora bajt  z PAO
LD   C, (IX-5)    ;załaduj do rejestru C  bajt  z PAO
LD   (IX+11), A   ;zapamiętaj rejestr A w komórce PAO
LD   (IY),    H   ;zapamiętaj rejestr H w komórce PAO
LD   (IX-1),  33  ;wpisz  do komórki PAO  stałą 33
ADD  A, (IX-20)   ;do akumulatora dodaj bajt z PAO
```

Rejestry IX i IY można ładować i zapamiętać w PAO, układać i zdejmować ze stosu, wymieniać z wierzchołkiem stosu, inkrementować i dekrementować, dodawać do nich zawartość innej pary rejestrów oraz podwajać, podobnie jak parę HL:

```
LD   IX, 12345      ;wpisz stałą do rejestru IX
LD   IY, (20000)    ;do IY skopiuj słowo z PAO
LD   (20000), IX    ;zapamiętaj w PAO zawartość IX
PUSH IY             ;przechowaj IY na stosie
EX   (SP), IX       ;przechowaj IY na stosie
INC  IY             ;inkrementuj IY bez zmiany bitów stanu
DEC  IX             ;dekrementuj IX bez zmiany bitów stanu
ADD  IX,  BC        ;do IX dodaj zawartość pary BC
ADD  IY,  IY        ;podwój zawartość IY
```

Rozkazy ADD IX,.... i ADD IY,... wpływają tylko na bit CY (jak ADD HL,...). Procesor Z80 dopuszcza rozkazy ADC i SBC także z udziałem HL:

```
ADC  HL,  BC     ;dodaj z przeniesieniem BC do HL
SBC  HL,  DE     ;odejmij  z pożyczką DE  od  HL
```

Dopuszczalne argumenty: BC, DE, HL i SP. W odróżnieniu od ADD HL,... rozkazy te wpływają jednak na wszystkie bity stanu, podobnie jak ADD A,... Chcąc odjąć parę rejestrów od HL bez uwzględniania pożyczki, wystarczy przed rozkazem SBC HL,... wyzerować CY, chociażby przez AND A.

Z80 ma specjalne rozkazy do operacji na blokach PAO. Rozkazy LDI i LDD pozwalają przesyłać bajty między komórkami PAO bez udziału akumulatora. HL zawiera adres źródłowy, DE — docelowy. Po przesłaniu w LDD pary HL i DE są dekrementowane, w LDI — inkrementowane, zaś para BC jest dekrementowana zarówno przez LDD, jak i przez LDI. Rozkazy LDDR i LDIR działają jak LDD i LDI, lecz są automatycznie powtarzane do chwili wyzerowania zawartości BC. Oba rozkazy pozwalają w prosty sposób rozwiązać przemieszczanie dużych bloków PAO. Oto przykład przesyłania 1000 bajtów spod adresu 20000 pod 30000:

```
LD   HL, 20000    ;adres pocz. obszaru źródłowego do HL
LD   DE, 30000    ;adres pocz. obszaru docelowego do DE
LD   BC, 1000     ;licznik bajtów do BC
LDIR              ;prześlij blok
```

CPD, CPI, CPIR i CPDR służą do przeszukiwania obszarów PAO. Porównują one komórkę wskazaną przez HL z zawartością A, a następnie dekrementują BC oraz dekrementują (CPD, CPDR) albo inkrementują (CPI, CPIR) parę HL. CPIR i CPDR są przy tym powtarzane automatycznie do chwili wyzerowania pary BC lub napotkania w PAO bajtu zgodnego z zawartością A. Przyczynę zakończenia powtórzeń wskazują bity stanu: Z=1, gdy wykryto zgodność, P/V=1, gdy został wyzerowany rejestr BC. CY pozostaje niezmieniony.

Z80 pozwala testować, kasować lub ustawiać pojedyncze bity dowolnego rejestru roboczego lub komórki PAO:

```
BIT  1, A        ;testuj bit nr 1 w akumulatorze
RES  0, H        ;seruj  bit nr 0 w rejestrze  H
SET  6, (HL)     ;ustaw  bit nr  6 w komórce PAO
BIT  7, (IX+2)   ;testuj najstarszy bit komórki PAO
```

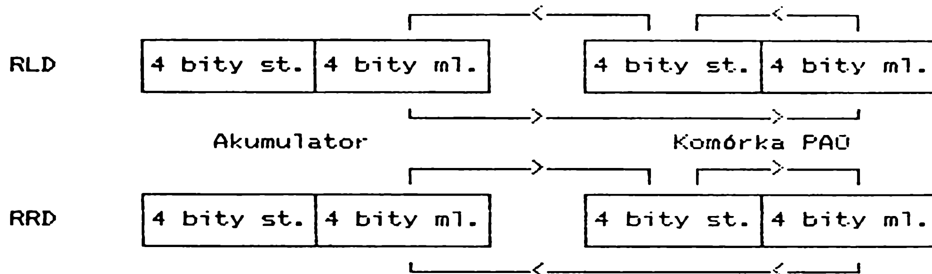
RES i SET nie zmieniają bitów stanu. BIT wpływa tylko na Z. Z=1, gdy testowany bit =0. Z80 umożliwia rotację dowolnych rejestrów i komórek PAO:

```
RL  A         ;rotuj w lewo  akumulator    (jak RLA)
RR  C         ;rotuj w prawo rejestr C     (jak RRA)
RLC (HL)      ;rotuj w lewo  komórkę PAO (jak RLCA)
RRC (IX-5)    :rotuj w prawo komórkę PAO (jak RRCA)
```

Choć działanie jest podobne, jak w przypadku omówionych wcześniej rozkazów rotacji akumulatora, rozkazy te zmieniają wszystkie bity stanu w zależności od wyniku przesunięcia. Oprócz rotacji możliwe są przesunięcia: SLA (arytmetyczne w lewo), SRA (arytmetyczne w prawo) oraz SRL (logiczne w prawo). Możliwe argumenty — j.w. Opuszczający rejestr lub komórkę PAO bit przechodzi zawsze do CY, lecz wolny bit z drugiego końca wypełniany jest zerem (SLA i SRL) lub zachowuje poprzednią zawartość (SRA). Dzięki temu rozkaz SRA może służyć do dzielenia przez 2 liczb ze znakiem (bit znaku jest powielany). Rozkazy przesunięć także wpływają na wszystkie bity stanu:

```
SLA  B        ;przesuń arytmetycznie w lewo treść B
SRA  (HL)     ;przesuń arytmet. w prawo komórkę PAO
SRL  (IY+7)   ;przesuń logicznie w lewo komórkę PAO
```

Istnieją dwa specyficzne rozkazy rotacji, operujące na półbajtach i przydatne zwłaszcza przy operacjach na liczbach formatu BCD: RLD i RRD:



Komórka PAO jest wskazywana zawartością pary HL, starszy półbajt akumulatora pozostaje niezmieniony. Zmienione są wszystkie bity stanu prócz CY.

Z80 pozwala wymieniać informacje między portami wejścia/wyjścia, a dowolnym rejestrem procesora. Numer portu musi się przy tym znajdować w rejestrze C:

```
IN   A, (C)    ;wprowadź do A bajt z podanego portu
IN   L, (C)    ;wprowadź do C bajt z podanego portu
OUT  (C), A    ;wpisz do podanego portu zawartość A
OUT  (C), D    :wpisz do podanego portu zawartość D
```

W odróżnieniu od IN A, (STAŁA), rozkaz IN A, (C) zmienia wartości bitów stanu odpowiednio do wprowadzonej wartości. Rozkazy IND, INI, OUTD i OUTI przesyłają bajt między portem o numerze w rejestrze C oraz komórką PAO wskazywaną przez HL. Po przesłaniu zawartość HL jest dekrementowana (IND i OUTD) lub inkrementowana (INI i OUTI), zaś rejestr B (nie para BCI) jest dekrementowany. Rozkazy INDR, INIR, OTDR i OTIR działają podobnie, lecz są powtarzane automatycznie aż do wyzerowania rejestru B.

7. ZAMIAST ZAKOŃCZENIA

Programowanie jest umiejętnością praktyczną, dlatego najlepiej opanować je, przeplatając lekturę podręczników ćwiczeniami przy komputerze. Programowania nie można nauczyć się wyłącznie przy pomocy ołówka i kartki papieru! Przy nauce programowania w języku asemblera niezbędna jest systematyczność i wytrwałość. Nie wolno zrażać się niepowodzeniami. Błędy popełniają nawet najlepsi programiści. Należy zawsze postępować tak, aby wykrycie błędu nie nastręczało większych kłopotów. Dlatego warto od początku narzucić sobie surową dyscyplinę w kwestii poprawnej struktury i dokumentacji programów — nawet wtedy, gdy będzie to służyło tylko wyrobieniu dobrych nawyków. Trzeba zawsze postępować tak, jak gdyby opracowane programy miały stanowić nasz kapitał na przyszłość. Nie należy porywać się na zbyt ambitne projekty, nie dysponując odpowiednim doświadczeniem. Kłopoty z uruchomieniem programu rosną o wiele szybciej, niż liczba instrukcji.

Planując złożone przedsięwzięcia programistyczne, trzeba przemyśleć przede wszystkim struktury danych i algorytmy manipulacji nimi. Decyzję o zastosowaniu konkretnego języka programowania lepiej odłożyć na później. Może okazać się bowiem, że użycie języka asemblera wcale nie jest konieczne. Nowoczesne języki programowania, jak C i PASCAL, rozsądnie użyte, dają znacznie więcej możliwości bezpośredniej współpracy ze sprzętem i optymalnego wykorzystania procesora, niż się wydaje na pierwszy rzut oka. Przy bliższym poznaniu mechanizmy te często okazują się zupełnie wystarczające. Aby je wykorzystać, potrzebna jest jak zwykle odpowiednia wiedza. Najlepszym jej źródłem są zazwyczaj pełne wersje oryginalnych, firmowych podręczników. Premiowana będzie przy tym zapewne znajomość języków obcych. Tłumaczenia, wykonywane przez pokątne tzw. studia komputerowe i inne przypadkowe, a nie budzące zaufania „instytucje", są zwykle mocno okrojone i roją się od błędów, także merytorycznych. Jeśli zawarte tam wiadomości okażą się niewystarczające, warto czasem pokusić się o samodzielną analizę fragmentu wyprodukowanego przez kompilator kodu. Kiedy nabierzemy ogólnego wyczucia, jaki kod maszynowy odpowiada jakim konstrukcjom w wersji źródłowej, poprzez celową budowę programu źródłowego możemy nieraz znacznie powiększyć jego efektywność. Znajomość języka maszynowego może zatem przydać się nawet wtedy, jeśli nie będziemy w nim bezpośrednio programować!

Oprócz prób samodzielnego układania programów dobrą szkołą jest też analiza gotowych, porządnie opracowanych programów. Można tam podejrzeć wiele interesujących technik programowania, a nawet sztuczek programistycznych. Sztuczek z reguły kopiować nie należy, ale często stanowią one ciekawą ilustrację specyficznych właściwości procesora i mogą stanowić inspirację dla własnej aktywności. Programy takie publikowane są w książkach i czasopismach, czasem można otrzymać też wersje źródłowe programów na nośnikach maszynowych.

W chwili obecnej język asemblera używany jest do dużych projektów prawie wyłącznie przez profesjonalistów. Użytkownicy posługują się zazwyczaj łatwiejszymi w użyciu języka-

mi wysokiego poziomu, programując w asemblerze tylko te fragmenty algorytmu, które wymagają szczególnej szybkości, odwołują się do mechanizmów sprzętowych lub z innych powodów nie dają się efektywnie zrealizować. Typowym przykładem może być obsługa przerwań zewnętrznych. Większość języków wysokiego poziomu umożliwia korzystanie z procedur napisanych w języku asemblera. Niekiedy podprogram maszynowy złożony z kilku zaledwie rozkazów potrafi zdziałać cuda (np. w grafice ekranowej). Przykładem mogą być wszelkiego rodzaju szybkie przemieszczenia zawartości zwartych bloków pamięci ekranu, wywołujące złudzenie płynnego ruchu.

Jest kilka powodów, dla których grafika komputerowa — zwłaszcza ekranowa — jest szczególną domeną języka asemblera. Typowa rozdzielczość graficzna komputerów osobistych waha się od 256*192 do 720*348 punktów. Daje to od ok. 49 tysięcy do ok. 250 tysięcy niezależnie adresowanych punktów ekranu, zaś pojemność pamięci ekranu wynosi — w przypadku obrazu monochromatycznego — od 6 do 31 KB. Jeśli obraz jest barwny, liczby te ulegną przynajmniej podwojeniu. Nawet uwzględniając fakt, że skonstruowanie rysunku lub przemieszczenie figury po ekranie wymaga modyfikacji tylko części tych punktów, toi tak zadanie stojące przed procesorem okaże się poważne. Nawet bardzo szybki procesor nie dysponuje w takim przypadku rezerwą wydajności. Wydłużenie czasu operacji na ekranie z 0,1 s do 0,2 s zdecydowanie pogarsza wrażenie.

Operacje w pamięci ekranu są z reguły dość prymitywne i często ograniczają się do kopiowania lub przesuwania bajtów, lub ustawiania i kasowania pojedynczych bitów. Stosując język asemblera i uwzględniając specyfikę listy rozkazów, a czasem i charakterystyczne cechy organizacji pamięci ekranu, można takie operacje zaprogramować bardzo efektywnie, osiągając czasem kilkakrotną oszczędność czasu w porównaniu z kodem maszynowym wyprodukowanym nawet przez dobry kompilator języka wysokiego poziomu. W przypadku standardowych operacji np. obliczeniowych przewaga asemblera jest znacznie mniej wyraźna.

Dla wielu znanych kompilatorów, jako np. TURBO — PASCAL, dostępne są biblioteki procedur maszynowych, realizujących np. typowe operacje graficzne. Właściwe i celowe użytkowanie tych procedur będzie o wiele łatwiejsze wtedy, gdy dobrze znamy możliwości procesora i potrafimy przewidzieć, kiedy zastosowanie języka maszynowego da odpowiedni efekt.

Znajomość języka maszynowego jest wręcz nieodzowna przy rozmaitych przeróbkach i adaptacjach oprogramowania systemowego. Współczesne systemy operacyjne, np. popularny w świecie ośmiobitowców CP/M 80, pozostawiają użytkownikowi rozliczne, zazwyczaj dobrze udokumentowane „furtki", pozwalające dopasować system do własnych potrzeb. Wachlarz możliwości jest szeroki: od zadań tak skomplikowanych, jak instalacja systemu operacyjnego na nietypowym sprzęcie, po względnie proste, jak np. dołączenie do systemu nowej drukarki. Potrzebnego oprogramowania najczęściej nie trzeba nawet tworzyć od podstaw. Przeważnie wystarczy posłużyć się jako wzorcem podobnymi rozwiązaniami, publikowanymi w literaturze fachowej lub w dokumentacji dostarczonej przez producenta. Często załatwią sprawę drobne poprawki, jak zmiany adresów pamięci, portów urządzeń wejścia/ wyjścia lub stałych czasowych.

Powyżej przedstawiono garść powodów, dla których warto znać przynajmniej podstawy programowania w języku maszynowym. Listę tę można by oczywiście jeszcze wydłużyć, nie o to jednak chodzi. Abstrahując od zastosowań profesjonalnych, programowanie w języku asemblera może być bowiem wspaniałym, twórczym hobby. Hobby dla każdego myślącego człowieka, pragnącego dogłębnie poznać, zrozumieć i ujarzmić mikrokomputer — kolejną wielką sensację XX wieku.

DODATEK A

Konwersja liczb między systemami: dwójkowym, szesnastkowym i dziesiętnym

Pionowa kolumna mieści cztery młodsze bity (młodszy półbajt) oraz odpowiadającą mu cyfrę szesnastkową, poziomy nagłówek — cztery starsze bity. Okna zawierają dziesiętną postać liczby dwójkowej, złożonej ze starszego i młodszego półbajtu odpowiadającego danej kolumnie i wierszowi. Tablica składa się z 3 części. Pierwsza zawiera liczby od 0 do 127, jednoznacznie interpretowane jako dodatnie. Druga obejmuje liczby dwójkowe z ustawionym najstarszym bitem dla przypadku, gdy bajt jest traktowany jako liczba bez znaku (128 — 255). Trzecia część tablicy zawiera liczby z ustawionym najstarszym bitem (bitem znaku) w przypadku, gdy są one interpretowane jako liczby ze znakiem w systemie dopełnienia dwójkowego, czyli liczby ujemne od –1 do –128. Przykłady:

11000011B =OC3H =195 lub, jeśli ze znakiem, –61
74 = 01001010B = 4AH –108 = 10010100B = 94H
3BH = 00111011B = 59

heks	heks bin.	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111
0	0000	0	16	32	48	64	80	96	112
1	0001	1	17	33	49	65	81	97	113
2	0010	2	18	34	50	66	82	98	114
3	0011	3	19	35	51	67	83	99	115
4	0100	4	20	35	52	68	84	100	116
5	0101	5	21	37	53	69	85	101	117
6	0110	6	22	38	54	70	86	102	118
7	0111	7	23	39	55	71	87	103	119
8	1000	8	24	40	56	72	88	104	120
9	1001	9	25	41	57	73	89	105	121
A	1010	10	26	42	58	74	90	106	122
B	1011	11	27	43	59	75	91	107	123
C	1100	12	28	44	60	76	92	108	124
D	1101	13	29	45	61	77	93	109	125
E	1110	14	30	46	62	78	94	110	126
F	1111	15	31	47	63	79	95	111	127

heks	heks bin.	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	0000	128	144	160	176	192	208	224	240
1	0001	129	145	161	177	193	209	225	241
2	0010	130	146	162	178	194	210	226	242
3	0011	131	147	163	179	195	211	227	243
4	0100	132	148	164	180	196	212	228	244
5	0101	133	149	165	181	197	213	229	245
6	0110	134	150	166	182	198	214	230	246
7	0111	135	151	167	183	199	215	231	247
8	1000	136	152	168	184	200	216	232	248
9	1001	137	153	169	185	201	217	233	249
A	1010	138	154	170	186	202	218	234	250
B	1011	139	155	171	187	203	219	235	251
C	1100	140	156	172	188	204	220	236	252
D	1101	141	157	173	189	205	221	237	253
E	1110	142	158	174	190	206	222	238	254
F	1111	143	159	175	191	207	223	239	255

heks	heks bin.	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	0000	-128	-112	-96	-80	-64	-48	-32	-16
1	0001	-127	-111	-95	-79	-63	-47	-31	-15
2	0010	-126	-110	-94	-78	-62	-46	-30	-14
3	0011	-125	-109	-93	-77	-61	-45	-29	-13
4	0100	-124	-108	-92	-76	-60	-44	-28	-12
5	0101	-123	-107	-91	-75	-59	-43	-27	-11
6	0110	-122	-106	-90	-74	-58	-42	-26	-10
7	0111	-121	-105	-89	-73	-57	-41	-25	-9
8	1000	-120	-104	-88	-72	-56	-40	-24	-8
9	1001	-119	-103	-87	-71	-55	-39	-23	-7
A	1010	-118	-102	-86	-70	-54	-38	-22	-6
B	1011	-117	-101	-85	-69	-53	-37	-21	-5
C	1100	-116	-100	-84	-68	-52	-36	-20	-4
D	1101	-115	-99	-83	-67	-51	-35	-19	-3
E	1110	-114	-98	-82	-66	-50	-34	-18	-2
F	1111	-113	-97	-81	-65	-49	-33	-17	-1

DODATEK B

Lista rozkazów maszynowych wspólnych dla 8080 i Z80

Lewa kolumna przedstawia te rozkazy Z80 (ułożone alfabetycznie), które mają odpowiedniki w rozkazach 8080. Obok przedstawiono notację owych rozkazów w konwencji INTEL. Rubryka „kod maszynowy" przedstawia kody rozkazów w postaci dziesiętnej i szesnastkowej. Pierwszym bajtem jest zawsze kod operacji: „nn" oznacza argument jednobajtowy, „nnn" — dwubajtowy (najpierw bajt młodszy, następnie starszy). CEnn oznacza rozkaz dwubajtowy o kodzie OCEH i jednobajtowym argumencie. Rubryka „Bity stanu" wskazuje, na które z czterech podstawowych wskaźników stanu wpływa dany rozkaz. „x" oznacza, że bit ustawiony jest zależnie od wyniku, „0" — bit jest zawsze zerowany, „1" — bit zawsze ustawiany, CY oznaczono jako C, P/V jako P.

Zapis w języku asemblera		Kod maszynowy		Bity stanu
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	C S Z P
ADC A, (HL)	ADC M	142	8E	x x x x
ADC A, A	ADC A	143	8F	x x x x
ADC A, B	ADC B	136	88	x x x x
ADC A, C	ADC C	137	89	x x x x
ADC A, D	ADC D	138	8A	x x x x
ADC A, E	ADC E	139	8B	x x x x
ADC A, H	ADC H	140	8C	x x x x
ADC A, L	ADC L	141	8D	x x x x
ADC A, nn	ACI nn	206	CEnn	x x x x
ADD A, (HL)	ADD M	134	86	x x x x
ADD A, A	ADD A	135	87	x x x x
ADD A, B	ADD B	128	80	x x x x
ADD A, C	ADD C	129	81	x x x x
ADD A, D	ADD D	130	82	x x x x
ADD A, E	ADD E	131	83	x x x x

Zapis w języku asemblera		Kod maszynowy		Bity stanu
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	C S Z P
ADD A, H	ADD H	132	84	x x x x
ADD A, L	ADD L	133	85	x x x x
ADD A, nn	ADI nn	198	C6nn	x x x x
ADD HL, BC	DAD B	9	09	x
ADD HL, DE	DAD D	25	19	x
ADD HL, HL	DAD H	41	29	x
ADD HL, SP	DAD SP	57	39	x
AND (HL)	ANA M	166	A6	0 x x x
AND A	ANA A	167	A7	0 x x x
AND B	ANA B	160	A0	0 x x x
AND C	ANA C	161	A1	0 x x x
AND D	ANA D	162	A2	0 x x x
AND E	ANA E	163	A3	0 x x x
AND H	ANA H	164	A1	0 x x x
AND L	ANA L	165	A2	0 x x x
AND nn	ANI nn	230	E6nn	0 x x x
CALL C , nnnn	CC nnnn	220	DCnnnn	
CALL M , nnnn	CM nnnn	252	FCnnnn	
CALL NC, nnnn	CNC nnnn	212	D4nnnn	
CALL NZ, nnnn	CNZ nnnn	196	C4nnnn	
CALL P , nnnn	CP nnnn	244	F4nnnn	
CALL PE, nnnn	CPE nnnn	236	ECnnnn	
CALL PO, nnnn	CPO nnnn	228	E4nnnn	
CALL Z , nnnn	CZ nnnn	204	CCnnnn	
CALL nnnn	CALL nnnn	205	CDnnnn	
CCF	CNC	63	3F	x
CP (HL)	CMP M	190	BE	x x x x
CP A	CMP A	191	BF	x x x x
CP B	CMP B	184	B8	x x x x
CP C	CMP C	185	B9	x x x x
CP D	CMP D	186	BA	x x x x

Zapis w języku asemblera		Kod maszynowy		Bity stanu C S Z P
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	
CP E	CMO E	187	BB	x x x x
CP H	CMP H	188	BC	x x x x
CP L	CMP L	189	BD	x x x x
CP nn	CPI nn	254	FFnn	x x x x
CPL	CMA	39	2F	
DAA	DAA	47	27	x x x x
DEC (HL)	DCR M	53	35	x x x
DEC A	DCR A	61	3D	x x x
DEC B	DCR B	5	05	x x x
DEC BC	DCX B	11	0B	
DEC C	DCR C	13	0D	x x x
DEC D	DCR D	21	15	x x x
DEC DE	DCX D	27	1B	
DEC E	DCR E	29	1D	x x x
DEC H	DCR H	37	25	x x x
DEC HL	DCX H	43	2B	
DEC L	DCR L	45	2D	x x x
DEC SP	DCX SP	59	3B	
DI	DI	243	F3	
DI	EI	251	FB	
EX (SP),HL	XTHL	227	E3	
EX DE, HL	XCHG	235	EB	
HALT	HLT	118	76	
IN A,(nn)	IN nn	219	DBnn	
INC (HL)	INR M	52	34	x x x
INC A	INR A	60	3C	x x x
INC B	INR B	4	04	x x x
INC BC	INX B	3	03	
INC C	INR C	12	0C	x x x
INC D	INR D	20	14	x x x
INC DE	INX D	19	13	

Zapis w języku asemblera		Kod maszynowy		Bity stanu
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	C S Z P
INC E	INR E	28	1C	x x x
INC H	INR H	36	24	x x x
INC HL	INX H	35	23	
INC L	INR L	44	2C	x x x
INC SP	INX SP	51	33	
JP (HL)	PCHL	233	E9	
JP C, nnnn	JC nnnn	218	DAnnnn	
JP M, nnnn	JM nnnn	250	FAnnnn	
JP NC,nnnn	JNC nnnn	210	D2nnnn	
JP NZ,nnnn	JNZ nnnn	194	C2nnnn	
JP P, nnnn	JP nnnn	242	F2nnnn	
JP PE,nnnn	JPE nnnn	234	EAnnnn	
JP PO,nnnn	JPO nnnn	226	E2nnnn	
JP Z, nnnn	JZ nnnn	202	CAnnnn	
JP nnnn	JMP nnnn	195	C3nnnn	
LD (BC), A	STAX B	2		
LD (DE), A	STAX D	18	12	
LD (HL), A	MOV M,A	119	77	
LD (HL), B	MOV M, B	112	70	
LD (HL), C	MOV M, C	113	71	
LD (HL), D	MOV M, D	114	72	
LD (HL), E	MOV M, E	115	73	
LD (HL), H	MOV M, H	116	74	
LD (HL), L	MOV M, L	117	75	
LD (HL), nn	MVI M, nn	54	36mm	
LD (nnnn), A	STA nnnn	50	32nnnn	
LD (nnnn), HL	SHLD nnnn	34	22nnnn	
LD A, (BC)	LDAX B	10	0A	
LD A, (DE)	LDAX D	26	1A	
LD A, (HL)	MOV A, M	126	7E	
LD A,(nnnn)	LDA nnnn	58	3Annnn	

Zapis w języku asemblera		Kod maszynowy		Bity stanu C S Z P
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	
LD A, A	MOV A, A	127	7F	
LD A, B	MOV A, B	120	78	
LD A, C	MOV A, C	121	79	
LD A, D	MOV A, D	122	7A	
LD A, E	MOV A, E	123	7B	
LD A, H	MOV A, H	124	7C	
LD A, L	MOV A, L	125	7D	
LD A, nn	MVI A, nn	62	3Enn	
LD B, (HL)	MOV B, M	70	46	
LD B, A	MOV B, A	71	47	
LD B, B	MOV B, B	64	40	
LD B, C	MOV B, C	65	41	
LD B, D	MOV B, D	66	42	
LD B, E	MOV B, E	67	43	
LD B, H	MOV B, H	68	44	
LD B, L	MOV B, L	69	45	
LD B, nn	MVI B, nn	6	06nn	
LD BC,nnnn	LXI B,nnnn	1	01nnnn	
LD C, (HL)	MOV C, M	78	4E	
LD C, A	MOV C, A	79	4F	
LD C, B	MOV C, B	72	48	
LD C, C	MOV C, C	73	49	
LD C, D	MOV C, D	74	4A	
LD C, E	MOV C, E	75	4B	
LD C, H	MOV C, H	76	4C	
LD C, L	MOV C, L	77	4D	
LD C, nn	MVI C, nn	14	0Enn	
LD D, (HL)	MOV D, M	86	56	
LD D, A	MOV D, A	87	57	
LD D, B	MOV D, B	80	50	
LD D, C	MOV D, C	81	51	

Zapis w języku asemblera		Kod maszynowy		Bity stanu C S Z P
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	
LD D, D	MOV D, D	82	52	
LD D, E	MOV D, E	83	53	
LD D, H	MOV D, H	84	54	
LD D, L	MOV D, L	85	55	
LD D, nn	MVI D, nn	22	16nn	
LD DE,nnnn	LXI D,nnnn	17	11nnnn	
LD E, (HL)	MOV E, M	94	5E	
LD E, A	MOV E, A	95	5F	
LD E, B	MOV E, B	88	58	
LD E, C	MOV E, C	89	59	
LD E, D	MOV E, D	90	5A	
LD E, E	MOV E, E	91	5B	
LD E, H	MOV E, H	92	5C	
LD E, L	MOV E, L	93	5D	
LD E, nn	MVI E, nn	30	1Enn	
LD H, (HL)	MOV H, M	102	66	
LD H, A	MOV H, A	103	67	
LD H, B	MOV H, B	96	60	
LD H, C	MOV H, C	97	61	
LD H, D	MOV H, D	98	62	
LD H, E	MOV H, E	99	63	
LD H, H	MOV H, H	100	64	
LD H, L	MOV H, L	101	65	
LD H, nn	MVI H, nn	38	26nn	
LD HL,(nnnn)	LHLD nnnn	2A	42nnnn	
LD HL,nnnn	LXI H,nnnn	33	21nnnn	
LD L, (HL),	MOV L, M	110	6E	
LD L, A	MOV L, A	111	6F	
LD L, B	MOV L, B	104	68	
LD L, C	MOV L, C	105	69	
LD L, D	MOV L, D	106	6A	

Zapis w języku asemblera		Kod maszynowy		Bity stanu C S Z P
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	
LD L, E	MOV L, E	107	6B	
LD L, H	MOV L, H	108	6C	
LD L, L	MOV L, L	109	6D	
LD L, nn	MVI L, nn	46	2Enn	
LD SP, HL	SPHL	249	F9	
LD SP,nnnn	LXI SP,nnnn	49	31nnnn	
NOP	NOP	0	00	
OR (HL)	ORA M	182	B6	0 x x x
OR A	ORA A	183	B7	0 x x x
OR B	ORA B	176	B0	0 x x x
OR C	ORA C	177	B1	0 x x x
OR D	ORA D	178	B2	0 x x x
OR E	ORA E	179	B3	0 x x x
OR H	ORA H	180	B4	0 x x x
OR L	ORA L	181	B5	0 x x x
OR nn	ORI nn	246	F6nn	0 x x x
OUT (nn), A	OUT nn	121	D3nn	
POP AF	POP PSW	241	F1	x x x x
POP BC	POP B	193	C1	
POP DE	POP D	209	D1	
POP HL	POP H	225	E1	
PUSH AF	PUSH PSW	245	F5	
PUSH BC	PUSH B	197	C5	
PUSH DE	PUSH D	213	D5	
PUSH HL	PUSH H	229	E5	
RET	RET	201	C9	
RET C	RC	216	D8	
RET M	RM	248	F8	
RET NC	RNC	208	D0	
RET NZ	RNZ	192	C0	
RET P	RP	240	F0	

Zapis w języku asemblera		Kod maszynowy		Bity stanu
ZILOG Z80	INTEL 8080	dziesiętny	szesnastkowy	C S Z P
RET PE	RPE	232	E8	
RET PO	RPO	224	E0	
RET Z	RZ	200	C8	
RLA	RAL	23	17	x
RLCA	RLC	7	07	x
RRA	RAR	31	1F	x
RRCA	RRC	15	OF	x
RST O	RST O	199	C7	
RST 8	RST 1	207	CF	
RST 16	RST 2	215	D7	
RST 24	RST 3	223	DF	
RST 32	RST 4	231	E7	
RST 40	RST 5	239	FF	
RST 48	RST 6	247	F:	
RST 56	RST 7	255	FF	
SBC A, (HL)	SBB M	158	9E	x x x x
SBC A, A	SBB A	159	9F	x x x x
SBC A, B	SBB B	152	98	x x x x
SBC A, C	SBB C	153	99	x x x x
SBC A, D	SBB D	154	9A	x x x x
SBC A, E	SBB E	155	9B	x x x x
SBC A, H	SBB H	156	9C	x x x x
SBC A, L	SBB L	157	9D	x x x x
SBC A, nn	SBI nn	222	DEnn	x x x x
SCF	STC	55	37	1
SUB (HL)	SUB M	150	96	x x x x
SUB A	SUB A	151	97	x x x x
SUB B	SUB B	144	90	x x x x
SUB C	SUB C	145	91	x x x x
SUB D	SUB D	146	92	x x x x
SUB E	SUB E	147	93	x x x x

Zapis w języku asemblera		Kod maszynowy		Bity stanu C S Z P
ZILOG Z80	INTEL 8080	dziesiętnie	szesnastkowo	
SUB H	SUB H	148	94	x x x x
SUB L	SUB L	149	95	x x x x
SUB nn	SUI nn	214	D6nn	x x x x
XOR (HL)	XRA M	174	AE	0 x x x
XOR A	XRA A	175	AF	0 x x x
XOR B	XRA B	168	A8	0 x x x
XOR C	XRA C	169	A9	0 x x x
XOR D	XRA D	170	AA	0 x x x
XOR E	XRA E	171	AB	0 x x x
XOR H	XRA H	172	AC	0 x x x
XOR L	DRA L	173	AD	0 x x x
XOR nn	XRI nn	238	EEnn	0 x x x

DODATEK C

Odpowiedzi na pytania w tekście

1. LD A,45 ładuje do A stałą 45, zaś LD A, (45) — liczbę z komórki PAO o adresie 45. Zapis: LD 250, A jest bezsensowny — nie można przypisać zawartości akumulatora stałej liczbowej; taki rozkaz nie może istnieć!

2. Za pośrednictwem akumulatora:

```
LD   A, (20000)
LD   L, A
LD   A, (20001)
LD   H, A
```

3. Rozkaz ADD A,A mnoży zawartość akumulatora przez 2 i przenosi do CY najstarszy bit pierwotnej zawartości akumulatora.

4. Oto arytmetyczna negacja A:

```
CPL
INC  A
```

5. Zerowanie akumulatora można osiągnąć także rozkazem XOR A lub SUB A (odjęcie akumulatora od siebie daje 0).

6. Oto możliwe rozwiązanie:

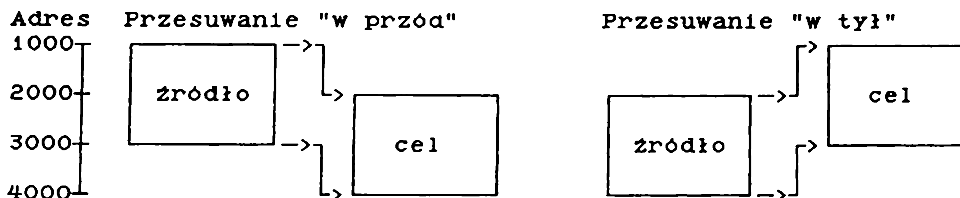
```
LD   A, (30000)
AND  A
JP   Z, ZER
```

   Nie wystarczy tylko załadować zawartości komórki do A, gdyż bity stanu nie zmienią przy tym stanu. Trzeba przeto wykonać na zawartości A jakąkolwiek operację arytmetyczną lub logiczną, nie zmieniającą jego zawartości, np. AND A, OR A, ADD O itd.

7. Liczba 0 jest uważana za liczbę dodatnią (bit nr 7 =0). Dlatego gdy wynik=0, nastąpi skok w rozkazie JP P,..., a więc skok JP Z nie ma szans wystąpić. Trzeba zamienić miejscami oba rozkazy skoku.

8. Nie można! Dodawanie rejestrów B i C może dać wynik zerowy także wówczas, gdy zawartość pary BC, rozumianej jako rejestr szesnastobitowy, nie jest zerem. Niech rejestr B zawiera FD, zaś rejestr C — 03. FD może być interpretowane zarówno jako 253, jak i −3. W wyniku dodawania powstaje przeniesienie, lecz akumulator zawiera 0, zaś bit warunku Z jest ustawiony. Po zastąpieniu rozkazu OR C przez ADD C pętla wykonałaby się zaledwie 3 razy!

9. Kierunek ma znaczenie wtedy, gdy obszary źródłowy i docelowy zachodzą na siebie. Ma to miejsce np. przy przesuwaniu dużej grupy bajtów na niewielką odległość:
Adres przesuwanie „w przód" przesuwanie „w tył"



Przesuwając blok pamięci „w przód", tzn, w kierunku większych adresów, musimy zacząć od ostatnich komórek. Ponieważ początkowy fragment obszaru docelowego pokrywa się z końcowym fragmentem obszaru źródłowego. Przy kopiowaniu od początku do końca zostałaby zniszczona nieprzeniesiona jeszcze część źródła. Przepisując od końca, wpisujemy nową zawartość do komórek, których pierwotna treść została już wcześniej przepisana. Przy przesuwaniu bloku PAO „w tył" trzeba zacząć od pierwszych komórek.

10. Najprościej odjąć od zera zawartość pary HL: −HL=0−HL:

```
XOR  A       ;zeruj akumulator
SUB  L       ;odejmij  od 0 młodszy bajt
LD   L, A    ;zapisz młodszy bajt wyniku
LD   A, 0    ;zeruj A bez zmiany bitów stanu
SBC  A, H    ;odejmij  od 0 starszy bajt
LD   H, A    ;zapisz starszy bajt wyniku
```

11. 17 HL=16 HL+HL. Dodając parę HL do samej siebie podwajamy jej zawartość. Czterokrotne powtórzenie tej operacji odpowiada mnożeniu przez 16:

```
LD   C,  L   ;skopiuj pierwotną zawaratość pary
LD   B,  H   ;do pary rejestrów BC    (BC= HL)
ADD  HL,HL   ;podwój zawartość HL     (HL= 2*HL)
ADD  HL,HL   ;podwój zawartość HL     (HL= 4*HL)
ADD  HL,HL   ;podwój zawartość HL     (HL= 8*BC)
ADD  HL,HL   ;podwój zawartość HL     (HL=16*BC)
ADD  HL,BC   ;do HL dodaj BC          (HL=17*BC)
```

12. Jeśli sprawdzanie, czy BC=0, będzie odbywać się przed pierwszym dodawaniem, wynik będzie poprawny:

```
MNOZ2:  LD   HL,  0
MNP:    LD   A,   C
        OR   B
        JP   Z,   KONIEC
        ADD  HL,  DE
        DEC   BC
        JP   MNP
KONIEC: ................
```

13. Wystarczy zastąpić LD BC, 0 przez LD BC, −1 (LD BC, OFFFFH).

14. Zamiana zawartości BC i DE: (zauważmy, że POP DE zdejmie ze stosu wartości ułożone przez PUSH BC!)

```
PUSH  BC
PUSH  DE
POP   BC
POP   DE
```

15. Za pośrednictwem pary HL:

```
LD   HL, 0
ADD  HL, SP
LD   (500), HL
```

16. Podprogram nie zdejmuje ze stosu przechowanej tam zawartości HL. Wskutek tego rozkaz RET zamiast adresu powrotu napotka na wierzchołku stosu pierwotną treść HL i skok nastąpi do komórki o adresie odpowiadającym tej zawartości.

17. Jeśli tylko możemy swobodnie korzystać z pary HL, to rozkaz RET da się zastąpić przez:

```
POP  HL
JP   (HL)
```

18. Oto rozwiązanie bez JP: (rozkaz: LD A, STAŁA nie wpływa na bity stanu, ustawione po AND A)

```
SIGN: AND  A
      RET  Z
      LD   A, -1
      RET  M
      LD   A,  1
      RET
```

19. RRA można zamienić przez RRCA — jego zadaniem jest tylko wpisywanie kolejnych bitów akumulatora do CY. Zastąpienie RLA przez RLCA sprawiłoby, że do najstarszej pozycji akumulatora byłby wprowadzany nie bit CY, lecz najmłodszy bit A.

20. Oto propozycja modyfikacji:

```
LD   A, '0'
ADC  A, 0
CALL PISZZN
```

Jeśli CY=0, to rozkaz ADC A, 0 nie zmieni zawartości A, w przeciwnym razie zwiększy ją o 1, co oznacza zmianę kodu '0' na '1'.

21. Można, stosując rekursje. W podprogramie WSPAK wystarczy tylko zamienić miejscami obydwa rozkazy CALL!

22. Aby podprogram WYSDEC wyświetlał liczby dwójkowe, wystarczy zamienić LD DE, 10 na LD DE, 2. Gdyby trzeba wyświetlać liczby szesnastkowe, to poza zamianą LD DE, 10 na LD DE, 16 trzeba zapewnić rozpoznanie cyfr większych niż 9:

```
      POP  AF         ;odtwórz w A resztę z dzielenia
      CP   10         ;czy większa lub równa 10 ?
      JP   C,  CYF    ;nie - cyfra "0" - "9"
CYF:  ADD  A,  7      ;różnica między 'A' i '9'+1 = 7
      ADD  A,  '0'    ;oblicz kod odpowiedniej  cyfry
```

23. Po pierwsze HL trzeba mnożyć nie przez 10, lecz przez 16, zastępując ADD HL, BC przez ADD HL, HL (rozkazy: LD C, L i LD B, H są zbędne). Po drugie, należy rozpoznawać cyfry szesnastkowe i poprawnie określać ich wartość:

```
DECBI1: LD    A, (DE)    ;odczytaj z bufora kolejny znak
        CP    'F'+1      ;czy kod większy od cyfry "F" ?
        RET   P          ;jeśli tak, to nie cyfra szesn.
        SUB   '0'        ;odejmij kod cyfry "0"
        RET   M          ;jeśli wynik  <0,  to nie cyfra
        CP    10         ;czy liczba w A większa od  9 ?
        JP    M,CYFRA    ;jeśli nie,   cyfra  dziesiętna
        SUB   7          ;7 = różnica między 'A' i '9'+1
        CP    10         ;gdy  A>=10, cyfra szesnastkowa
        RET   M          ;to nie jest cyfra szesnastkowa
 CYFRA: LD    C, A       ;..................
```

Studencka Oficyna Wydawnicza ZSP „Alma-Press"
Warszawa 1988
Wydanie I
Nakład 39 650 + 350 egz.
Ark. wyd. 5. Ark. druk. 5
Skład wykonały Prasowe Zakłady Graficzne w Ciechanowie
na urządzeniach fotoskładu CR-200
Podpisano do druku w kwietniu 1988 r.
Druk ukończono w styczniu 1989 r.
Druk i oprawę wykonały Prasowe Zakłady Graficzne
w Bydgoszczy, ul. Dworcowa 13; zam. 1674/88 K-18

ISBN 83 7020 041 9

Cena zł 300,-

almapress